
3ª EDIÇÃO
Revisada e Ampliada
INTRODUÇÃO À PROGRAMAÇÃO COM
PYTHON
Algoritmos
e lógica de programação
para iniciantes
novatec
Nilo Ney Coutinho Menezes

INTRODUÇÃO À PROGRAMAÇÃO COM

# PYTHON

Algoritmos e lógica de programação para iniciantes

**3ª Edição**

INTRODUÇÃO À PROGRAMAÇÃO COM

# PYTHON

Algoritmos e lógica de programação para iniciantes

**3ª Edição**

**Nilo Ney Coutinho Menezes**

Novatec

Editor: Rubens Prates
Revisão gramatical: Tássia Carvalho
Editoração eletrônica: Camila Kuwabata
Capa: Victor Bittow

ISBN: 978-85-7522-718-3

Histórico de impressões:

| | |
|---|---|
| Julho/2019 | Primeira reimpressão |
| Janeiro/2019 | Terceira edição (ISBN: 978-85-7522-718-3) |
| Junho/2014 | Segunda edição (ISBN: 978-85-7522-408-3) |
| Novembro/2010 | Primeira edição (ISBN: 978-85-7522-250-8) |

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
E-mail: novatec@novatec.com.br
Site: novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec
LIS20190723

Distribuição mundial (exceto Brasil):
Coutinho Menezes Nilo – LogiKraft
Rue de la Grande Campagne, 40
7340 Wasmes
Belgium
+32 485 251460
livros@logikraft.be

*A minha esposa, Chris; e aos meus filhos, Igor, Hanna e Iris.*

# Sumário

# Agradecimentos

Este livro não seria possível sem o incentivo de minha esposa, Emília Christiane, e a compreensão de meus filhos, Igor, Hanna e Iris. Para quem mora em um país frio como a Bélgica, escrever no verão nem sempre é fácil.

Também agradeço o suporte oferecido por meus pais e avós durante meus estudos, e os conselhos, a compreensão e a orientação que sempre recebi.

Não poderia me esquecer de meu irmão, Luís Victor, pela ajuda com as imagens em português.

A Luciano Ramalho e aos colegas da lista python-brasil. Luciano, obrigado pelo incentivo para publicar este livro e pelos comentários mais que pertinentes. À comunidade python-brasil pelo esforço e prova de civilidade ao manter as discussões em altíssimo nível, agradando a iniciantes, curiosos e profissionais de computação.

Agradecimentos também aos colegas, amigos e alunos da Fundação Matias Machline, onde tive oportunidade de estudar e trabalhar como professor de lógica de programação. Aos amigos e colegas do Centro Educacional La Salle e da Fundação Paulo Feitoza, onde ministrei cursos de lógica de programação e de Python.

# Prefácio da terceira edição

O sucesso da linguagem Python como primeira linguagem de programação não para de crescer. Cada vez mais, a linguagem é adotada em cursos de computação, engenharia e mesmo em outras áreas. As vantagens da sintaxe clara e das inúmeras bibliotecas abriram as portas da computação a setores antes inacessíveis a não programadores.

Quando escrevi a primeira edição deste livro, Python ainda era um nicho para os iniciados da computação, mas crescia enormemente. A finalidade desta obra sempre foi ensinar os fundamentos da programação de computadores, mas também fornecer um conteúdo que será útil após a fase de introdução e que poderá ser desenvolvido durante toda a carreira de programação do leitor. Nesta terceira edição, detalhes da linguagem foram adicionados, mesmo aqueles que vão além dos objetivos de um curso de lógica de programação, mas que são essenciais nas atividades de um profissional iniciante.

Cada capítulo foi revisado de forma a mostrar as vantagens das últimas versões da linguagem. Várias seções também foram adicionadas de modo a proporcionar melhor contextualização do conteúdo apresentado. Assim, o objetivo da terceira edição é aumentar a familiaridade do leitor com a linguagem Python e prepará-lo para tópicos mais avançados.

Modificações grandes no site que acompanham este livro também estão em andamento. A ideia principal é aumentar a interação com os leitores e oferecer um local controlado para tirar dúvidas, fazer críticas e sugestões.

Espero que essas modificações ajudem os leitores iniciantes, assim como aqueles que descobrem a linguagem Python por meio deste livro.

# Prefácio da segunda edição

Revisar este livro foi um trabalho muito gratificante, especialmente pelas mensagens de apoio recebidas na lista Python-Brasil e por email. Esta nova edição traz um capítulo extra, cobrindo o básico de banco de dados, no qual introduzimos o uso do SQLite. O Capítulo 10, que trata da orientação a objetos, também foi expandido, e vários conceitos específicos ao Python foram adicionados. Muitas pequenas revisões foram feitas por todo o livro, desde a atualização para Python 3.4 até a migração para Windows 8.1.

Apesar da tentação de cobrir tudo que se possa imaginar sobre Python, este livro continua uma obra dedicada àqueles que dão seus primeiros passos na programação de computadores. Nesta nova edição, conceitos mais específicos do Python também foram abordados para tornar o texto e nossos programas mais "Pythonicos".

# Prefácio da primeira edição

Aprendi a programar usando a linguagem BASIC ainda em meados dos anos 1980. Lembro-me de construir pequenos programas de desenho, agendas telefônicas e jogos. Armazenamento só em fitas K-7. Antes da internet, e morando no Norte do Brasil, a forma de aprender a programar era lendo livros e, claro, programando. A maneira mais comum de obter novos programas era por meio de revistas de programação, as quais eram verdadeiramente dedicadas à nova comunidade de usuários de microcomputadores, termo usado na época para diferenciar os computadores domésticos. Elas traziam listagens completas de programas, escritos em BASIC ou *Assembly*. Em uma época em que o download mal era um sonho distante, digitar esses programas era a única solução para executá-los. A experiência de ler e digitar os programas foi muito importante para aprender a programar, mas, infelizmente, poucas revistas técnicas hoje são acessíveis a iniciantes. A complexidade dos programas de hoje também é muito maior, exigindo mais tempo de estudo do que antigamente. Embora a internet ajude muito, seguir uma ordem planejada de aprendizado é muito importante.

Ao começar a ensinar programação, o desafio sempre foi encontrar um livro que pudesse ser lido por alunos do ensino médio ou no início do ensino superior. Embora várias obras tenham suprido essa necessidade, o uso de apostilas sempre foi necessário, pois a ordem em que os novos conceitos eram apresentados era quase sempre mais interessante para um dicionário do que para o ensino de programação em si. Conceitos importantes para o iniciante eram completamente esquecidos, e o foco maior era dado a assuntos mais complexos. Seguindo um conceito apresentado e utilizado por meus professores do ensino médio, a lógica de programação é mais importante do que qualquer linguagem. Quem aprende a programar uma vez fica mais apto a aprender outras linguagens de programação. É nessa lógica de programação para iniciantes que este livro se concentra, apresentando recursos do Python sempre que possível. O intuito é iniciar o leitor no mundo da programação e prepará-lo para cursos e conceitos mais avançados. Acredito que, depois de ler e estudar este livro, você estará apto a ler outras obras de programação e a aprender novas linguagens por conta própria.

# Introdução

Este livro foi escrito com o iniciante em programação em mente. Embora a linguagem Python seja muito poderosa e rica em recursos modernos de programação, esta obra não pretende ensinar apenas a linguagem em si, mas ensinar a programar em qualquer linguagem. Alguns recursos da linguagem Python não foram utilizados ou têm sua apresentação adiada para facilitar os exercícios de lógica de programação. O objetivo foi privilegiar esses exercícios e melhor preparar o leitor para outras linguagens. Essa escolha não impediu que recursos poderosos da linguagem fossem apresentados, mas este livro não é uma referência da linguagem Python.

Os capítulos foram organizados de forma a apresentar progressivamente os conceitos básicos de programação. A leitura do livro, feita perto de um computador com o interpretador Python aberto, é extremamente recomendada para facilitar a experimentação dos exemplos propostos. Alguns leitores obtiveram sucesso com interpretadores Python instalados em seus telefones celulares.

Cada capítulo traz exercícios organizados de modo a explorar o conteúdo apresentado. Alguns exercícios apenas modificam os exemplos do livro, enquanto outros requerem a aplicação dos conceitos apresentados na criação de novos programas. Tente resolver os exercícios na medida em que são apresentados. Embora seja impossível não falar de matemática em um livro de programação, os exercícios foram elaborados para o nível de conhecimento de um aluno do ensino médio, e utiliza problemas comerciais ou do dia a dia. Essa escolha não foi feita para evitar o estudo de matemática, mas para não misturar a introdução de conceitos de programação com novos conceitos matemáticos.

É recomendado que você organize os programas gerados, com uma pasta (diretório) por capítulo, de preferência adicionando o número do exemplo ou exercício aos nomes dos arquivos. Alguns exercícios alteram outros, mesmo em capítulos diferentes. Uma boa organização desses arquivos vai facilitar seu trabalho de estudo.

O Apêndice A foi preparado para ajudar a entender as mensagens de erro que podem ser geradas pelo interpretador Python. Utilize-o sempre que encontrar novos erros em seus programas. Com a prática, você aprenderá a reconhecer esses erros e a localizá-los em seus programas.

O uso de Python também libera alunos e professores para utilizar o sistema operacional de sua escolha, seja Windows, Linux ou Mac OS X. Todos os exemplos do livro requerem apenas a distribuição padrão da linguagem, que é disponibilizada gratuitamente.

Embora todo esforço tenha sido realizado para evitar erros e omissões, não há garantias de que o livro esteja isento de erros. Se você encontrar falhas no conteúdo, envie um email para `erros@nilo.pro.br`. Em caso de dúvidas, embora eu não possa garantir resposta a todos os emails, envie sua mensagem para `duvidas@nilo.pro.br`. Dicas, críticas e sugestões podem ser enviadas para `professores@nilo.pro.br`. O código-fonte e eventuais correções deste livro podem ser encontrados no site `http://python.nilo.pro.br`.

Um resumo do conteúdo de cada capítulo é apresentado a seguir:

**Capítulo 1 – Motivação**: visa a apresentar o desafio de aprender e estimular o estudo da programação de computadores, apresentando problemas e aplicações do dia a dia.

**Capítulo 2 – Preparação do ambiente**: instalação do interpretador Python, introdução ao editor de textos, apresentação do IDLE, ambiente de execução, como digitar programas e fazer os primeiros testes com operações aritméticas no interpretador.

**Capítulo 3 – Variáveis e entrada de dados**: tipos de variáveis, propriedades de cada tipo, operações e operadores. Apresenta o conceito de programa no tempo e uma técnica simples de rastreamento. Entrada de dados pelo teclado, conversão de tipos de dados e erros comuns.

**Capítulo 4 – Condições**: estruturas condicionais, conceitos de blocos e seleção de linhas a executar com base na avaliação de expressões lógicas.

**Capítulo 5 – Repetições**: estrutura de repetição `while`, contadores, acumuladores. Apresenta o conceito de repetição da execução de um bloco e de repetições aninhadas.

**Capítulo 6 – Listas, dicionários, tuplas e conjuntos**: operações com listas, ordenação pelo método de bolhas, pesquisa, utilização de listas como pilhas e filas. Exemplos de uso de dicionários, tuplas e conjuntos.

**Capítulo 7 – Trabalhando com strings**: apresenta operações avançadas com strings. Explora a classe string do Python. O capítulo traz também um jogo simples para fixar os conceitos de manipulação de strings.

**Capítulo 8 – Funções**: noção de função e transferência de fluxo, funções recursivas, funções lambda, parâmetros, módulos. Apresenta números aleatórios.

**Capítulo 9 – Arquivos**: criação e leitura de arquivos em disco. Geração de arquivos HTML em Python, operações com arquivos e diretórios, parâmetros pela linha de comando, caminhos, manipulação de datas e horários.

**Capítulo 10 – Classes e objetos**: introdução à orientação a objetos. Explica os conceitos de classe, objetos, métodos e herança. Prepara o aluno para continuar estudando o tópico e melhor compreender o assunto.

**Capítulo 11 – Banco de dados**: introdução à linguagem SQL e ao banco de dados SQLite.

**Capítulo 12 – Próximos passos**: capítulo final, que lista os próximos passos em diversos tópicos, como jogos, sistemas web, programação funcional, interfaces gráficas e bancos de dados. Visa a apresentar livros e projetos open source pelos quais o aluno pode continuar estudando, dependendo de sua área de interesse.

**Apêndice A – Mensagens de erro**: explica as mensagens de erro mais frequentes em Python, mostrando suas causas e como resolvê-las.

**Apêndice B – Adaptações para outras versões do Python**: mostra pequenas adaptações que devem ser feitas no código deste livro para que seja compatível com versões anteriores à 3.6.

# CAPÍTULO 1

# Motivação

*Então, você quer aprender a programar?*

Programar computadores é uma tarefa que exige tempo e dedicação para ser corretamente aprendida. Muitas vezes não basta só estudar e fazer os exemplos, mas também deixar a mente se acostumar com a nova forma de pensar. Para muitas pessoas, o mais difícil é continuar gostando de programar. Elas desistem nas primeiras dificuldades e não voltam mais a estudar. Outras são mais pacientes, aprendem a não se irritar com a máquina e a assumir seus erros.

Para não sofrer dos males de quem não aprendeu a programar, você precisa responder a algumas perguntas antes de começar:

1. Você quer aprender a programar?
2. Como está seu nível de paciência?
3. Quanto tempo você pretende estudar?
4. Qual é o seu objetivo ao programar?

## 1.1 Você quer aprender a programar?

Responda a essa questão, mas pense um pouco antes de chegar à resposta final. A maneira mais difícil de aprender a programar é não querer programar. A vontade deve vir de você, e não de um professor ou de um amigo. Programar é uma arte e precisa de dedicação para ser dominada. Como tudo o que é desconhecido, é muito difícil quando não a entendemos, mas se torna mais simples à medida que a aprendemos.

Se você já decidiu aprender a programar, siga para a próxima parte. Se ainda não se convenceu, continue lendo. Programar pode tornar-se um novo *hobby* e até mesmo uma profissão. Se você estuda computação, precisa saber programar. Isso não significa que você será um programador por toda a vida, ou que a programação limitará seu crescimento dentro da área de informática. Uma desculpa que já ouvi muitas vezes é "*eu sei programar, mas não gosto*". Vários alunos de computação terminam os cursos sem saber programar; isto é, sem realmente saber programar.

Programar é como andar de bicicleta, você não se esquece, mas só aprende fazendo. Ao trocar de uma linguagem de programação para outra, se você realmente aprendeu a programar, terá pouca dificuldade para aprender a nova linguagem. Diferentemente de saber programar, a sintaxe de uma linguagem de programação é esquecida muito facilmente. Não pense que saber programar é decorar todos aqueles comandos, parâmetros e nomes estranhos. Programar é saber utilizar uma linguagem de programação para resolver problemas, ou seja, saber expressar uma solução por meio de uma linguagem de programação.

## 1.2 Como está seu nível de paciência?

Seja paciente.

Outro erro de quem estuda programação é querer fazer coisas difíceis logo de início.

Qual será seu primeiro programa? Um editor de textos? Uma planilha eletrônica? Uma calculadora?

Não! Será algo bem mais simples... Como somar dois números.

É isso mesmo: somar dois números!

Com o tempo, a complexidade e o tamanho dos programas aumentarão.

Seja paciente.

Programar exige muita paciência e, principalmente, atenção a detalhes. Uma simples vírgula no lugar de um ponto ou aspas esquecidas podem arruinar seu programa. No início, é comum perder a calma ou mesmo se desesperar até aprender a ler o que realmente escrevemos em nossos programas. Nessas horas, paciência nunca é demais. Leia novamente a mensagem de erro ou pare para entender o que não está funcionando corretamente. Nunca pense que o computador está contra você, nem culpe o dia ou o destino.

Seja paciente.

## 1.3 Quanto tempo você pretende estudar?

Pode-se aprender a programar em poucas horas. Se você é o sujeito que programa o micro-ondas da sua tia, que abre vidros de remédio lendo as instruções da tampa ou que brincou de Lego, programar é seu segundo nome.

Todos nós já programamos algo na vida, nem que seja ir ao cinema no sábado. A questão é: quanto tempo você dedicará para aprender a programar computadores?

Como tudo na vida, nada de exageros. Na realidade, tanto o tempo como a forma de estudar variam muito de pessoa para pessoa. Algumas rendem mais estudando em grupo. Outras gostam de ter aulas.

O importante é incluir o estudo da programação em seu estilo preferido. Não tente aprender tudo ou entender tudo rapidamente. Se isso ocorrer, parabéns, há muito pela frente. Caso contrário, relaxe. Se não entender na segunda tentativa, pare e volte a tentar amanhã.

Quando encontrar um problema, tenha calma. Veja o que você escreveu. Verifique se entende o que está escrito. Um erro comum é querer programar sem saber ler as instruções. É como querer escrever sem saber falar. Você precisa entender o que faz um programa antes mesmo de executá-lo no computador. Esse entendimento é fundamental na programação. Utilize o computador apenas para tirar dúvidas e para praticar, nunca passe de um exercício para outro sem entender o que você fez ou porque tal operação foi realizada. Pergunte, anote suas dúvidas, não deixe para depois.

Inicie o estudo com sessões de uma ou no máximo duas horas por dia. Depois ajuste esse tempo a seu ritmo.

Aproveite a internet e as comunidades para conhecer outros programadores, aprender mais e também tirar dúvidas. Lembre-se de que você pode obter mais informações no site do livro e também enviar dúvidas por email.

## 1.4 Programar para quê?

Se você ainda não precisa programar para seu trabalho ou estudo, vejamos algumas outras razões:

### 1.4.1 Escrever páginas web

Hoje, todos estão expostos à web, à internet e a seus milhares de programas. A web só funciona porque permite a publicação de páginas e mais páginas de

textos e imagens com apenas um editor de textos. A mais complexa página que você já acessou é um conjunto de linhas de texto reunidas para instruir um programa, o navegador (browser), sobre como apresentar seu conteúdo.

### 1.4.2 Acertar seu relógio

Você conhece aquelas pessoas que nunca aprenderam a acertar seus relógios? Consigo me lembrar de várias delas...

Seguir instruções é muito importante para tarefas tão simples quanto essas. A sequência de passos para ajustar as horas, os minutos e até mesmo a data de seu relógio pode ser encarada como um programa. Normalmente, aperta-se o botão de ajuste até que um número comece a piscar. Depois, você pode usar um botão para mudar a hora ou ir direto para o ajuste dos minutos. Isso se repete até que você tenha ajustado todos os valores como segundos, dia, mês e, às vezes, o ano.

Se você nunca usou um relógio de pulso, tente encontrar um e ajustar as horas sozinho.

### 1.4.3 Aprender a usar mapas

Já se perdeu em uma cidade estranha? Já fez uma lista de passos para chegar a algum lugar? Então você já programou antes. Só de procurar um mapa você já mereceria um prêmio. Ao traçar um caminho de onde você está até onde deseja chegar, você normalmente relaciona uma lista de ruas ou marcos desse caminho. Normalmente é algo como passar três ruas à esquerda, dobrar à direita, dobrar à esquerda... Ou algo como seguir reto até encontrar um sinal de trânsito ou um rio. Isso é programar. Seguir seu programa é a melhor forma de saber se o que você escreveu está correto ou se você agora está realmente perdido.

Hoje, com GPS em nossos telefones, mapas em papel estão cada vez mais raros, mas a experiência é semelhante. Você consegue achar um caminho apenas escutando os comandos do GPS sem olhar o mapa? Ao obedecer às direções dadas pelo GPS, você está seguindo um programa, ou seja, uma sequência determinada de passos (instruções).

### 1.4.4 Mostrar para seus amigos que você sabe programar

Esta pode ser a razão mais complicada. Vamos ver isso como um subproduto do aprendizado, e não seu maior objetivo. Se essa for a razão de aprender a programar, é melhor continuar lendo e arranjar outra.

Programar é um esforço para você realizar algo. É uma tarefa que exige dedicação e que traz muita satisfação pessoal. Seus programas podem ser legais, mas aí já serão seus programas e não somente você.

### 1.4.5 Parecer estranho

Agora estamos conversando. Entender o que milhares de linhas de programa significam pode realmente gerar uma fama como essa entre os leigos. Se esse é seu objetivo, saiba que há maneiras mais fáceis, como parar de tomar banho, deixar as unhas crescerem, ter um cabelo de cor laranja ou roxa, parecer um roqueiro sem nunca ter tocado em uma banda etc.

Embora boa parte dos programadores que conheço não seja exatamente o que classifico de 100% normal, ninguém o é.

Saber programar não significa que você seja louco ou muito inteligente. Saber programar também não significa que você não seja louco ou que não seja muito inteligente. Imagine aprender a programar como qualquer outra coisa que você já aprendeu.

Um dia, sua mente poderá ter pensamentos estranhos, mas, quando esse dia chegar, pode apostar que você saberá.

### 1.4.6 Entender melhor como seu computador funciona

Programando você pode começar a entender por que aquela operação falhou ou por que o programa simplesmente fechou sem nem mesmo avisar o motivo.

Programar também pode ajudá-lo a utilizar melhor sua planilha ou editor de textos. O tipo de pensamento que se aprende programando servirá não só para fazer programas, mas também para usar programas.

### 1.4.7 Cozinhar

Uma vez precisei cozinhar um prato, mas as instruções estavam escritas em alemão. Não sei nada de alemão. Peguei o primeiro dicionário e comecei a traduzir as principais palavras. Com as palavras traduzidas, tentei entender o que deveria realmente fazer.

Naquela noite, o jantar foi apenas uma sopa instantânea. Receitas podem ser vistas como programas. E, como programas, só é possível segui-las se você entender aquilo que foi escrito.

A simples sequência de instruções não ajuda uma pessoa que não entenda seus efeitos.

Para algumas pessoas, programar é mais fácil que aprender alemão (ou qualquer outro idioma estrangeiro). E, como qualquer outra língua, não se aprende apenas com um dicionário.

Idiomas humanos são ricos em contextos, e cada palavra costuma ter múltiplos significados. A boa notícia: linguagens de programação são feitas para que máquinas possam entender o que ali foi representado. Isso significa que entender um programa é muito fácil, quase como consultar um dicionário. Outra boa notícia é que a maioria das linguagens contém conjuntos pequenos de "palavras".

Uma linguagem de programação possui uma capacidade de expressão limitada e um dos desafios de programar é justamente conseguir representar a solução de um problema vencendo essa limitação. Como o computador é uma máquina, essas linguagens são definidas e estruturadas de forma a serem traduzidas automaticamente em uma sequência de instruções que o computador consegue entender. Por isso, ao escrever seus programas, uma atenção redobrada ao que você escreve é muito importante, pois o computador não consegue interpretar intenções ou imaginar nada além do texto do programa.

### 1.4.8 Salvar o mundo

Uma boa razão para aprender a programar é salvar o mundo. Isso mesmo!

Todos os dias, milhares de quilos de alimento são desperdiçados ou não chegam aonde deveriam por falta de organização. Programando você pode ajudar a criar sistemas e até mesmo programas que ajudem outras pessoas a se organizar.

Outra boa ação é ajudar um projeto de software livre. Isso permitirá que muitas pessoas que não podem pagar por programas de computador se beneficiem deles sem cometer crime algum.

### 1.4.9 Software livre

Aliás, você tem licença de uso para todos os seus programas?

Se a resposta é não, saiba que programadores aprendem Linux e outros sistemas operacionais muito mais rapidamente. Também conseguem tirar maior proveito desses sistemas porque conseguem programá-los.

Se não existe, crie. Se é ruim, melhore.

Separar a programação em um mundo à parte pode ser sua primeira ideia estranha entre muitas.

Criar mundos dentro de programas e computadores pode ser a segunda.

## 1.5 Por que Python?

A linguagem de programação Python é muito interessante como primeira linguagem de programação devido à sua simplicidade e clareza. Embora simples, é também uma linguagem poderosa, podendo ser usada para administrar sistemas e desenvolver grandes projetos. É uma linguagem clara e objetiva, pois vai direto ao ponto, sem rodeios.

Python é software livre, ou seja, pode ser utilizada gratuitamente, graças ao trabalho da Python Foundation[1] e de inúmeros colaboradores. Você pode utilizar Python em praticamente qualquer arquitetura de computadores ou sistema operacional, como Linux[2], FreeBSD[3], Microsoft Windows ou Mac OS X[4].

Python vem crescendo em várias áreas da computação, como inteligência artificial, banco de dados, biotecnologia, animação 3D, aplicativos móveis (celulares), jogos e mesmo como plataforma web. Isso explica por que Python é famosa por ter "*batteries included*", ou seja, baterias inclusas, fazendo referência a um produto completo que pode ser usado prontamente (quem nunca ganhou um presente de Natal que veio sem pilhas?). Hoje é difícil encontrar uma biblioteca que não tenha *bindings* (versão) em Python. Esse fato torna o aprendizado da linguagem muito mais interessante, uma vez que aprender a programar em Python significa continuar a utilizar os conhecimentos adquiridos mesmo depois de aprender a programar para resolver problemas reais.

Uma grande vantagem de Python é a legibilidade dos programas escritos nessa linguagem. Outras linguagens de programação utilizam inúmeras marcações, como ponto (.) ou ponto e vírgula (;), no fim de cada linha, além dos marcadores de início e fim de bloco como chaves ({ }) ou palavras especiais (`begin/end`). Esses marcadores tornam os programas um tanto mais difíceis de ler e felizmente não são usados em Python. Veremos mais sobre blocos e marcações nos capítulos seguintes.

1 `http://www.python.org`

2 `http://www.kernel.org` ou `http://www.ubuntu.com` para obter o pacote completo

3 `http://www.freebsd.org`

4 `http://www.apple.com/macosx`

Outro bom motivo para aprender Python é obter resultados em pouco tempo. Como Python é uma linguagem completa, contando com bibliotecas para acessar bancos de dados, processar arquivos XML, construir interfaces gráficas e mesmo jogos, podemos utilizar muitas funções já existentes escrevendo poucas linhas de código. Isso aumenta a produtividade do programador, pois, ao utilizarmos bibliotecas, usamos programas desenvolvidos e testados por outras pessoas. Isso reduz o número de erros e permite que você se concentre realmente no problema que quer resolver.

Vejamos um pequeno programa escrito em Python.

```
print("Olá!")
```

Este programa tem apenas uma linha de código. A palavra **print** é uma função utilizada para enviar dados para a tela do computador. Ao escrevermos `print("Olá")`, ordenamos ao computador que exiba o texto "Olá!" na tela. Veja o que seria exibido na tela ao se executar esse programa no computador:

```
Olá!
```

Observe que as aspas (") não aparecem na tela. Esse é um dos detalhes da programação: precisamos marcar ou limitar o início e o fim de nossas mensagens com um símbolo, nesse caso, aspas. Como podemos exibir praticamente qualquer texto na tela, as primeiras aspas indicam o início da mensagem, e as seguintes, o fim. Ao programar, não podemos esquecer as limitações do computador. Um computador não interpreta textos como seres humanos. A máquina não consegue diferenciar o que é programa ou mensagem. Se não utilizarmos as aspas, o computador interpretará nossa mensagem como um comando da linguagem Python, gerando um erro.

O interpretador Python é uma grande ferramenta para o aprendizado da linguagem. O interpretador é o programa que permite digitar e testar comandos escritos em Python e verificar os resultados instantaneamente. Veremos como utilizar o interpretador na Seção 2.2.

A linguagem Python foi escolhida para este livro por simplificar o trabalho de aprendizado e fornecer grande poder de programação. Como é um software livre, disponível praticamente para qualquer tipo de computador, sua utilização não envolve a aquisição de licenças de uso, muitas vezes a um custo proibitivo.

Outra grande vantagem da linguagem Python é sua imensa comunidade de programadores e abundância de material disponível. Cada dia, mais pessoas aprendem a programar em Python e a linguagem se torna uma língua comum;

tudo que você vai aprender neste livro poderá ser usado durante sua carreira de programação. O fato de não precisar aprender outra linguagem para continuar trabalhando ou estudando também foi um dos maiores motivos para escolhê-la como linguagem.

Além disso, a sintaxe da linguagem Python é tão simples que o uso de uma pseudolinguagem, como Portugol, se torna desnecessário, facilitando o trabalho do aluno que codifica sua solução apenas uma vez e é capaz de executá-la em qualquer computador ou telefone celular.

Bem-vindo ao mundo da programação!

# CAPÍTULO 2

# Preparando o ambiente

Antes de começarmos, precisamos instalar o interpretador da linguagem Python. O interpretador é um programa que aceita comandos escritos em Python e os executa, linha a linha. É ele quem vai traduzir nossos programas em um formato que pode ser executado pelo computador. Sem o interpretador Python, nossos programas não podem ser executados, sendo considerados apenas como texto. O interpretador também é responsável por verificar se escrevemos corretamente nossos programas, mostrando mensagens de erro, caso encontre algum problema.

O interpretador Python não vem instalado com o Microsoft Windows: você deverá instalá-lo fazendo um download da internet. Se você utiliza Mac OS X ou Linux, provavelmente isso já foi feito, mas veja, a seguir, como verificar se você tem a versão correta.

## 2.1 Instalação do Python

Neste livro, usamos o Python versão 3.7. A linguagem sofreu diversas modificações entre as versões 2 e 3. A versão 3.7 foi escolhida por permitir a correta utilização dos novos recursos, além do fato de que é interessante aprendermos a versão mais nova. A versão 2.7 é também muito interessante, mas permite misturar a sintaxe (forma de escrever) do Python 2 com a do Python 3. Para evitar complicações desnecessárias, apresentaremos apenas as novas formas de escrever, e deixaremos o Python 3.7 verificar se fizemos tudo corretamente. O Python 3.7 ainda não estava disponível para todas as distribuições Linux na época de lançamento desta edição. Você pode utilizar também outras versões de Python 3 (de preferência 3.6 ou superior) com todos os exemplos do livro, embora pequenas alterações sejam necessárias nesses casos, vide Apêndice B.

### 2.1.1 Instalação no Windows

Python está disponível em todas as versões de Windows. Como é um software livre, podemos baixar o interpretador Python gratuitamente no site `http://www.python.org`. O site deve parecer com o da Figura 2.1. Veja que, no centro da página, temos a opção **Downloads**. Passe o mouse sobre **Downloads** e espere o menu abrir.

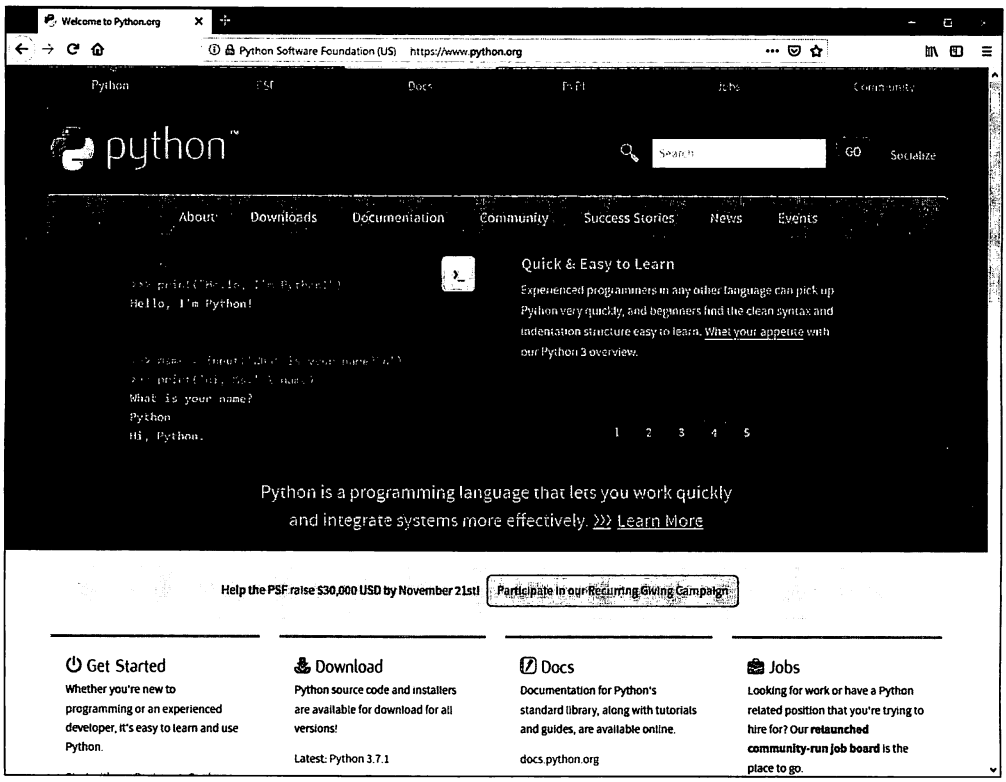


*Figura 2.1 – Site da Python Foundation.*

A Figura 2.2 mostra o menu da página de download aberto. Procure o texto Python 3.7.1. Se uma versão mais nova da série 3 estiver disponível, você deve escolhê-la. Quando este livro foi escrito, a versão mais nova era a 3.7.1. Clique no botão para iniciar o download do arquivo.

Se você não estiver utilizando o Firefox, o menu de download pode não aparecer. Nesse caso, clique em Windows e siga as instruções nas páginas que vão se abrir. Se você não conseguir encontrar esses arquivos, tente digitar o endereço completo da página de download: `https://www.python.org/downloads/release/python-370/`.

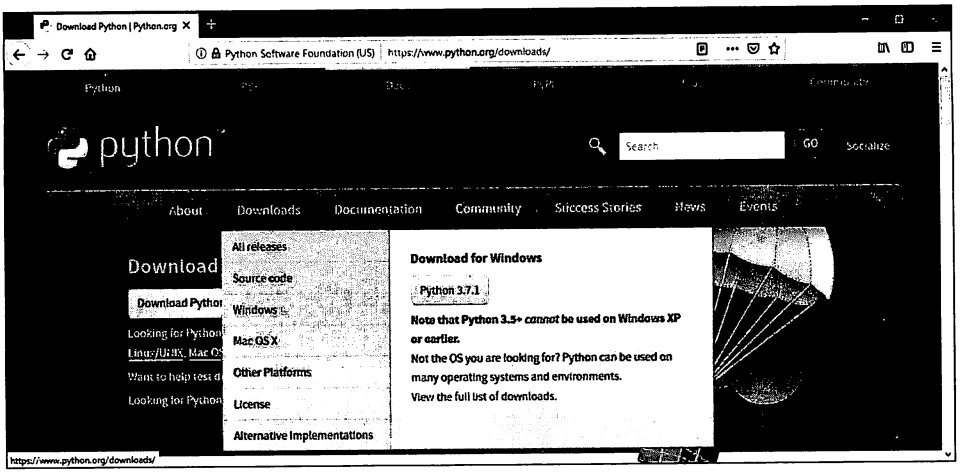


*Figura 2.2 – Página de download.*

Nos exemplos a seguir, faremos o download da versão 3.7.1 para Windows 10. A versão do Python pode mudar, e o nome do arquivo, também. Se você escolheu a versão de 32 ou 64 bits, o nome do arquivo será um pouco diferente, mas não se preocupe: se escolher a versão errada, ela simplesmente não funcionará. Você pode então tentar outra versão.

Clique no botão cinza com a versão do Python que você quer baixar, na Figura 2.2, clique em Python 3.7.1 para iniciar a transferência do arquivo.

A Figura 2.3 mostra a janela de download do Firefox. Essa janela também pode variar de acordo com a versão de seu navegador de internet. Clique no botão **Salvar arquivo** para iniciar a transferência do arquivo. Aqui, o Firefox foi utilizado como exemplo, mas você pode utilizar qualquer outro navegador de internet, como Internet Explorer, Google Chrome, Microsoft Edge ou Safari.

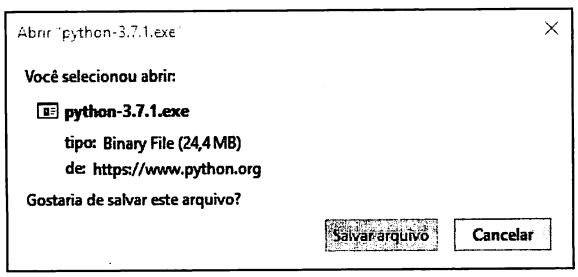


*Figura 2.3 – Janela de confirmação de download do Firefox.*

O interpretador Python ocupa aproximadamente 25 MB em disco (antes da instalação), e a transferência do arquivo deve demorar alguns minutos, dependendo da velocidade de conexão à internet. O Firefox exibe o download como na Figura 2.4.


*Figura 2.4 – O ícone de download muda de cor durante o download.*

Veja que a barra de transferência desaparece quando a transferência é concluída, como na Figura 2.5.

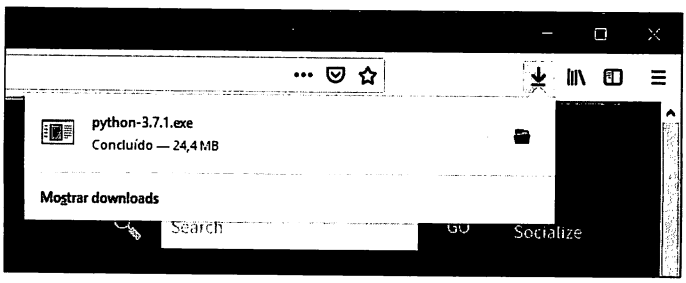


*Figura 2.5 – Janela de download com a transferência do arquivo finalizada.*

Clique no nome do arquivo que você acabou de baixar. Dependendo da versão do seu Windows e de seu navegador de internet, você poderá receber um pedido de confirmação do Firefox ou do Windows. Caso o Firefox pergunte se deseja executar o programa, escolha executar. O Windows também pode pedir uma confirmação de instalação, ilustrada na Figura 2.6. Escolha **Sim** para aprovar a instalação. Essa janela pode aparecer um pouco depois do início da instalação. Verifique também se um ícone parecido com um escudo colorido começar a piscar; nesse caso, clique no ícone para que a janela de permissão apareça.

A Figura 2.7 mostra a tela inicial do instalador. Até agora, não instalamos nada. É esse programa que vai instalar o Python 3.7 em seu computador. Clique **Customize installation** para continuar.

Sua janela deve se parecer com a da Figura 2.8. Selecione as opções como mostrado na figura e clique **Next**.

Você deve estar com uma janela parecida com a da Figura 2.9. Essa janela permite escolher onde instalar os arquivos do interpretador Python. Você pode mudar

a pasta ou simplesmente aceitar o local proposto (padrão). No exemplo, a pasta padrão foi alterada para `C:\python37-32` para facilitar o uso do interpretador na linha de comandos. Certifique-se que as opções de instalação em sua tela estão selecionadas como na figura.

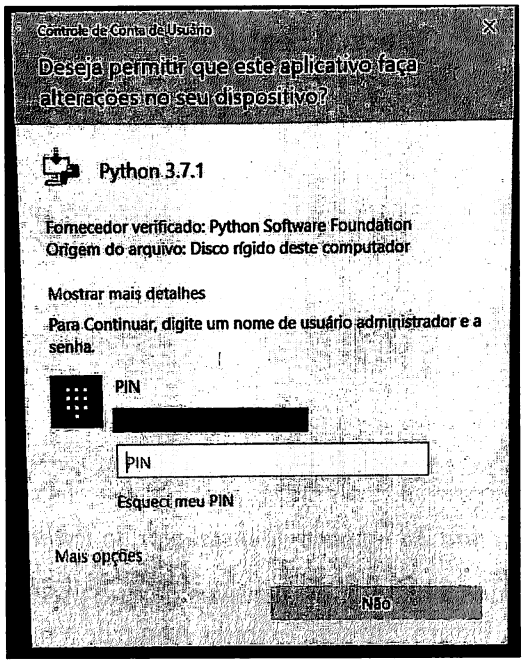


*Figura 2.6 – Janela de confirmação da execução do instalador Python.*

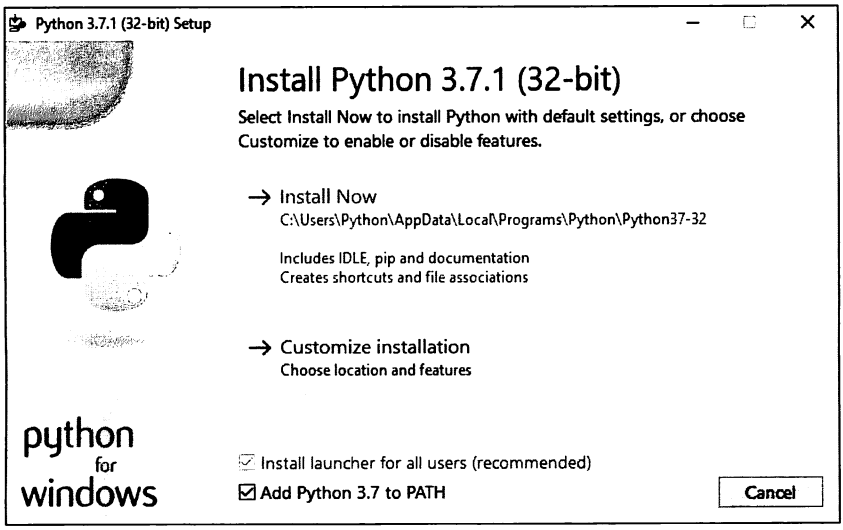


*Figura 2.7 – Janela do instalador do Python.*

A janela da Figura 2.9 permite selecionar que partes do pacote do Python queremos instalar. Essa janela é útil para utilizadores mais avançados da linguagem, mas vamos simplesmente aceitar o padrão, clicando no botão **Install** para continuar.

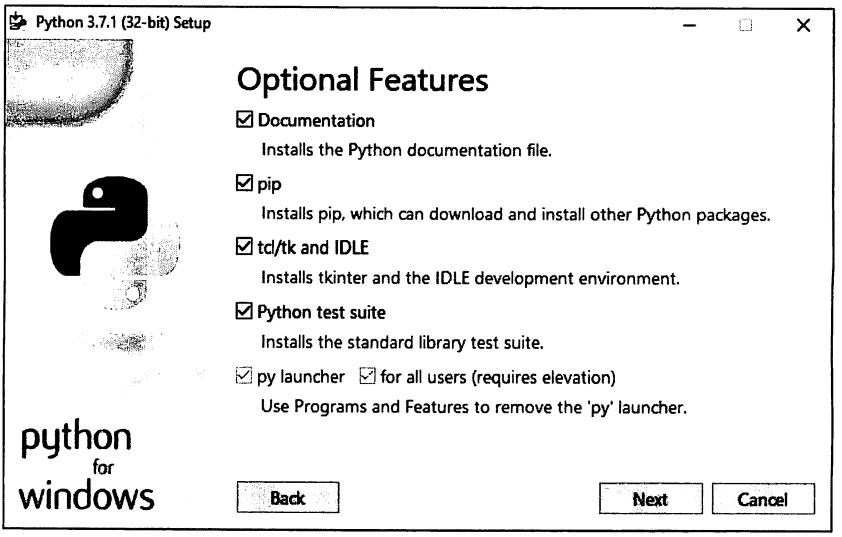


*Figura 2.8 – Janela do instalador do Python.*

Finalmente começamos a instalar os arquivos de que precisamos. Uma barra de progresso como a da Figura 2.10 será exibida. Aguarde a finalização do processo.

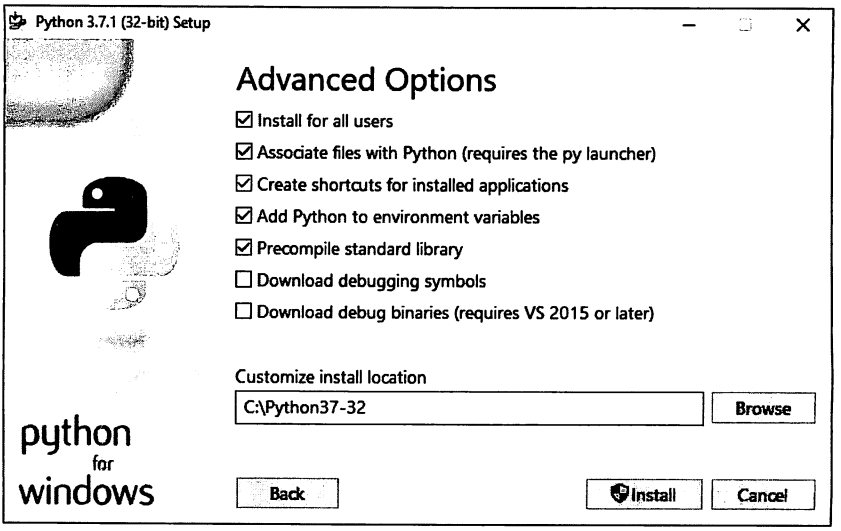


*Figura 2.9 – Janela de seleção das opções de instalação do Python.*

A instalação foi concluída. Clique no botão **Close**, mostrado na Figura 2.11, para terminar o instalador.

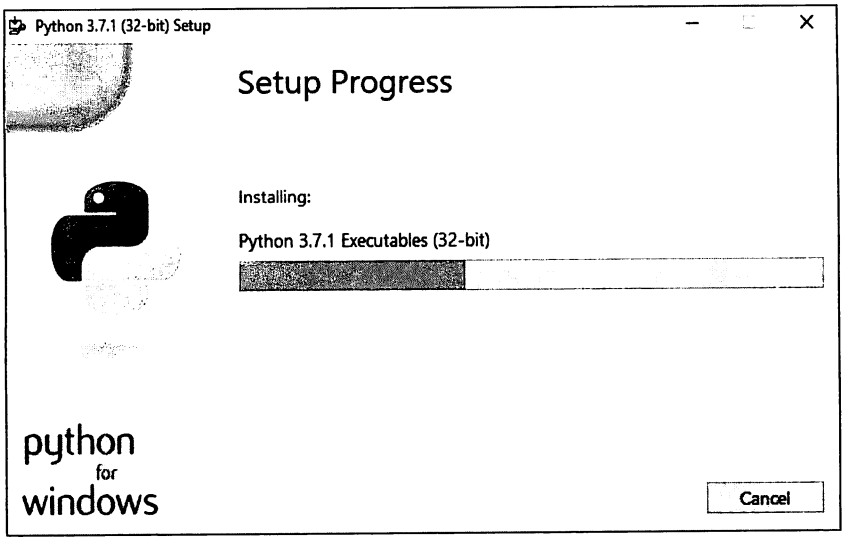


*Figura 2.10 – Janela do instalador exibindo o progresso da instalação.*

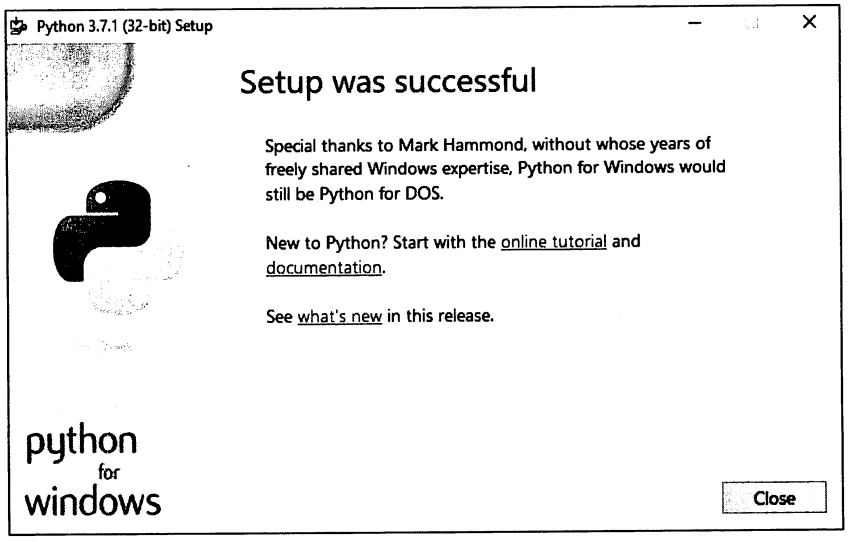


*Figura 2.11 – Janela de finalização do instalador.*

## 2.1.2 Instalação no Linux

Grande parte das distribuições Linux inclui uma versão do interpretador Python; porém, as versões mais utilizadas são a 2.7 ou a 3.5. Precisamos do interpretador na versão 3.6 ou superior. Para verificar a versão de seu interpretador, digite `python -V` na linha de comando.

No Ubuntu (versão 18.04), digite:

```
sudo apt update
sudo apt install -y python3.7 idle-python3.7
```

Você precisa de permissões de administrador (root) para efetuar a instalação. Se a versão 3.7 ainda não estiver disponível para instalação via apt-get, você pode utilizar a versão 3.6 sem problemas. Consulte o Apêndice B e o site do livro para mais detalhes.

### 2.1.3 Instalação no Mac OS X

Os Mac OS X vêm com uma versão do interpretador Python da Apple. No entanto, essa versão não é a 3.7. Para contornar o problema, instale o MacPorts (`http://www.macports.org/`), fazendo o download do arquivo `dmg`, e, depois, instale o Python 3.7 com:

```
sudo port install python37
```

Para executar o interpretador recém-instalado, digite `python3.7` na linha de comando. Você também pode utilizar outros gerenciadores de pacote disponíveis para Mac OS X, como Fink (`http://www.finkproject.org/`) e Homebrew (`http://brew.sh/`). Uma versão instalável, em formato `dmg`, também está disponível no site da `http://www.python.org`.

## 2.2 Usando o interpretador Python

Com o Python instalado, vamos começar a trabalhar.

O IDLE é uma interface gráfica para o interpretador Python, permitindo também a edição e execução de nossos programas. No Windows Vista e no Windows 7, você deve ter uma pasta no menu **Iniciar > Programas > Python 3.7**. Escolha **IDLE**. No Windows 10, procure por Python 3.7 na lista de aplicativos e, então, por IDLE (Python GUI).

No Linux, abra o terminal e digite:

```
idle-python3.7 &
```

No Mac OS X, abra o terminal e digite:

```
IDLE3.7 &
```

NOTA: se você não conseguiu instalar a versão 3.7 ou mais nova, adapte as linhas anteriores para a versão instalada, IDLE3.5 ou IDLE3.6, respectivamente para as versões 3.5 e 3.6.

A janela inicial do IDLE, no Windows 10, é mostrada na Figura 2.12. Se você utiliza o Mac OS X, Linux ou uma versão diferente de Windows, essa janela não será exatamente igual à da figura, mas muito parecida.

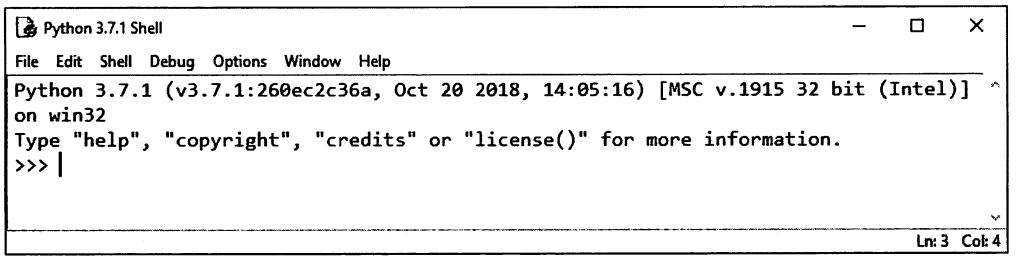


*Figura 2.12 – Janela inicial do IDLE.*

Observe que o cursor está posicionado dentro da janela Python Shell, e que a linha de comandos é iniciada pela sequência >>>.

Digite:

```
print("Oi")
```

Pressione a tecla **ENTER**. Você deve obter uma tela parecida com a da Figura 2.13.

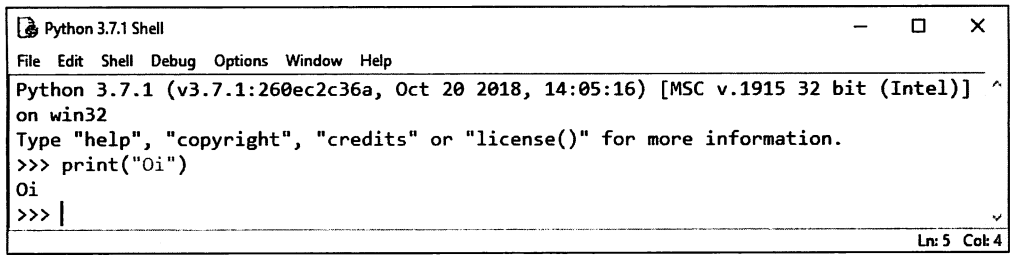


*Figura 2.13 – IDLE mostrando o resultado de print("Oi").*

Uma das grandes vantagens do Python é contar com o interpretador de comandos. Você pode abrir uma janela como essa sempre que tiver alguma dúvida em Python. O interpretador permite que você verifique o resultado de um comando instantaneamente. Veremos mais adiante que esse tipo de verificação é muito importante e facilita o aprendizado da linguagem.

## 2.3 Editando arquivos

Nem só de experimentos vive o programador Python. Um programa nada mais é que um arquivo-texto, escrito em um formato especial (linguagem).

No caso da linguagem Python, podemos usar qualquer editor de textos disponível: Notepad++ ou Sublime, no Windows; Sublime ou TextMate, no Mac OS X; Sublime, Vim ou Emacs, no Linux. Você também pode instalar o VSCode da Microsoft, distribuído gratuitamente, que funciona em Windows, Mac OS X e Linux. Embora bem pesado, o PyCharm também é bastante utilizado. Como a escolha do editor é muito pessoal, escolha o que mais gostar ou um que você já saiba utilizar.

O que você não deve utilizar é um editor de textos como o Microsoft Word ou o LibreOffice. Esses programas gravam seus arquivos em formatos especiais que não podem ser utilizados para escrever programas, salvo se você escolher gravar apenas texto ou simplesmente gravar no formato txt. Além disso, um editor de textos comum não foi feito para escrever programas.

Você pode também utilizar o editor de textos incluído na instalação do interpretador Python (IDLE). Com o interpretador aberto, clique no menu **File** e depois selecione a opção **New File**, como mostra a Figura 2.14.

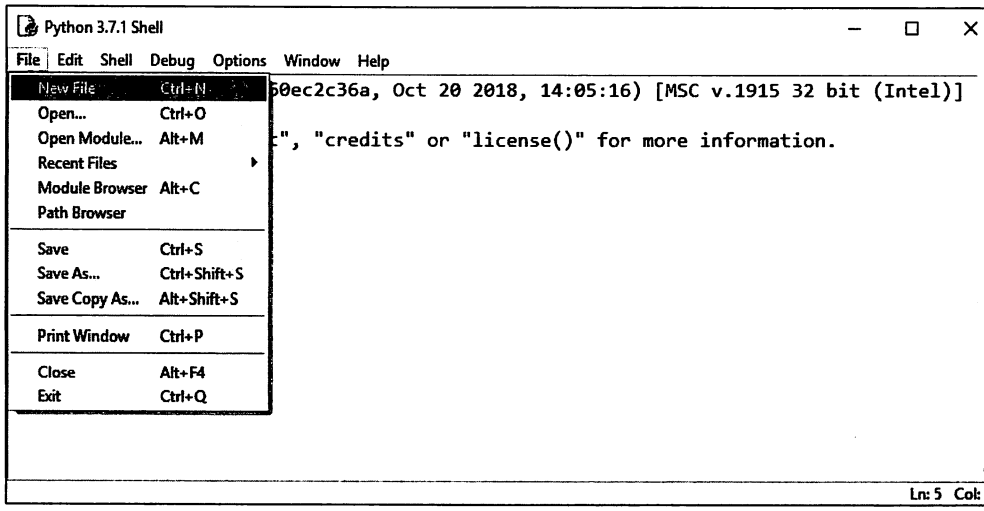


*Figura 2.14 – IDLE com o menu File aberto e a opção New File selecionada.*

Como utilizamos a versão 3.7 do interpretador, é muito importante que você use um editor de textos correto. Isto é, um editor que suporte a edição de textos em UTF-8. Se você não sabe o que é UTF-8, ou se o seu editor de textos suporta esse

formato de codificação, utilize o IDLE como editor de textos. Além de suportar a edição no formato UTF-8, que permitirá a utilização de acentos em nossos programas, ele é preparado para os colorir em Python, tornando a leitura mais fácil.

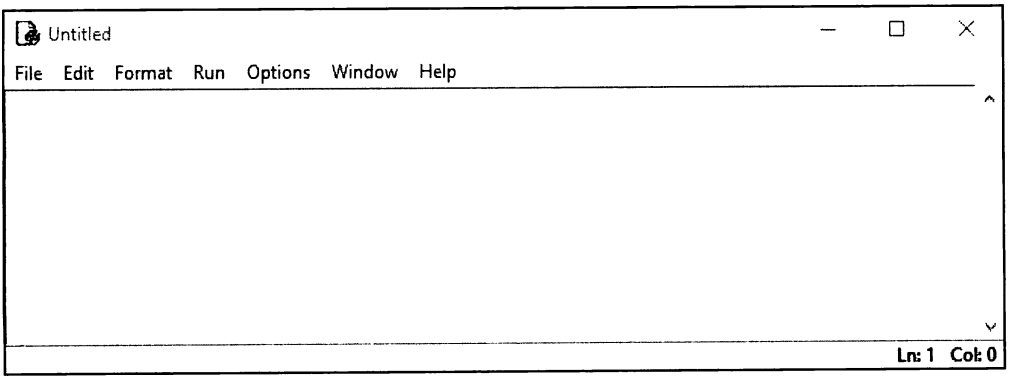


*Figura 2.15 – Janela do editor de textos do IDLE.*

Uma nova janela, como a da Figura 2.15, deverá aparecer. É nessa janela que escreveremos nossos programas. Observe que, embora parecida com a janela principal do IDLE, a janela tem opções de menu diferentes da outra. Para esclarecer essa separação, chamaremos a primeira de janela do interpretador; e a segunda, de janela do editor de textos. Se você ainda estiver em dúvida, a janela do editor de textos é a que apresenta a opção **Run** no menu.

Experimente um pouco, escrevendo:

```
print("Oi")
```

Sua janela do editor de textos deverá ser semelhante à mostrada na Figura 2.16.

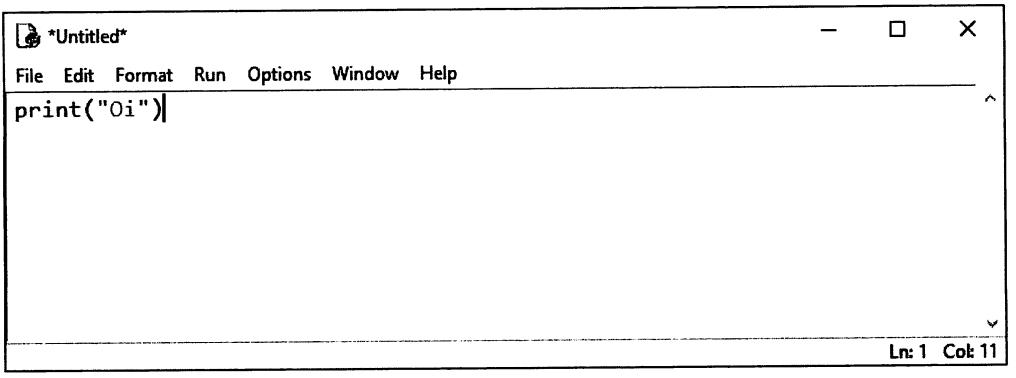


*Figura 2.16 – Editor de textos com o programa já digitado.*

Agora que escrevemos nosso pequeno programa, vamos salvá-lo. Escolha a opção **Save** do menu **File**, como mostra a Figura 2.17.

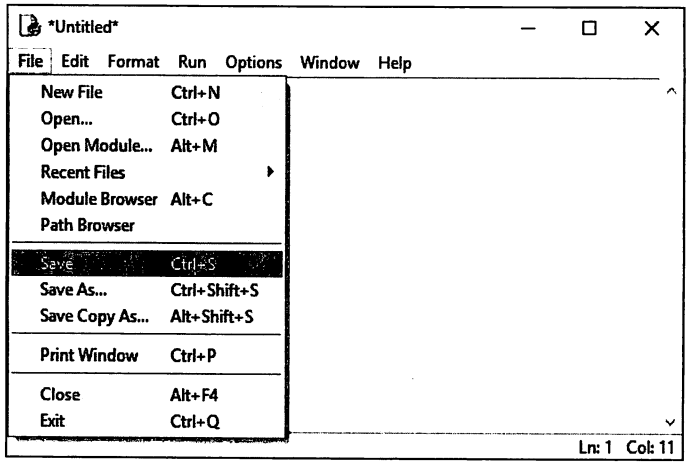


*Figura 2.17 – Janela do editor de textos com o menu File aberto e a opção Save selecionada.*

Uma janela-padrão de gravação de arquivos será exibida. A janela muda com a versão do sistema operacional, mas, se você utiliza Windows 10, ela parecerá com a da Figura 2.18. Escreva `teste.py` no nome do arquivo e clique no botão para salvar. Atenção: a extensão `.py` não é adicionada automaticamente pelo IDLE. Lembre-se de sempre gravar seus programas escritos em Python com a extensão `.py`.

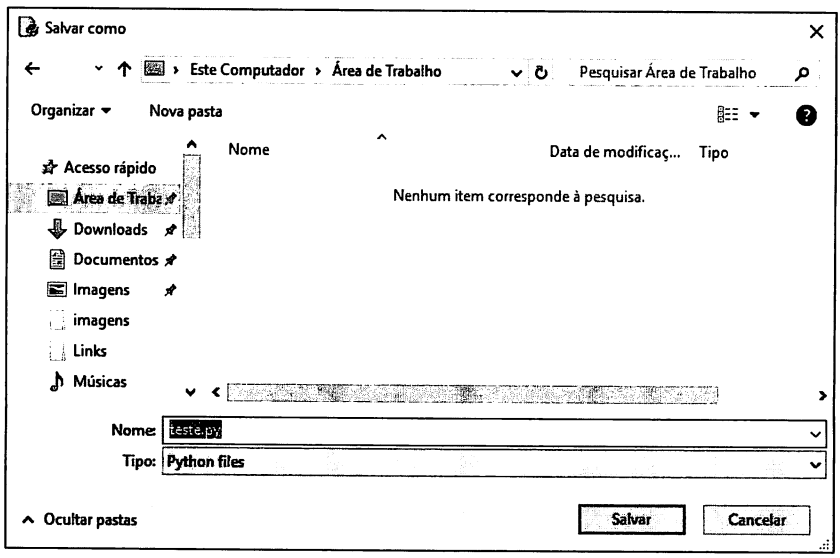


*Figura 2.18 – Janela de gravação de arquivos do Windows 10.*

Agora clique no menu **Run** para executar seu programa. Você pode também pressionar a tecla **F5** com o mesmo efeito. Essa operação pode ser visualizada na Figura 2.19.

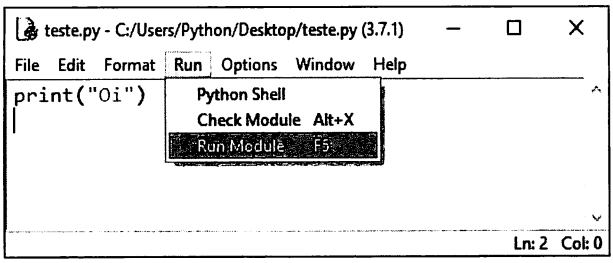


*Figura 2.19 – Janela do interpretador mostrando a opção Run Module selecionada.*

Seu programa será executado na outra janela, a do interpretador. Veja que uma linha com a palavra RESTART apareceu, como na Figura 2.20. Observe que obtivemos o mesmo resultado de nosso primeiro teste.

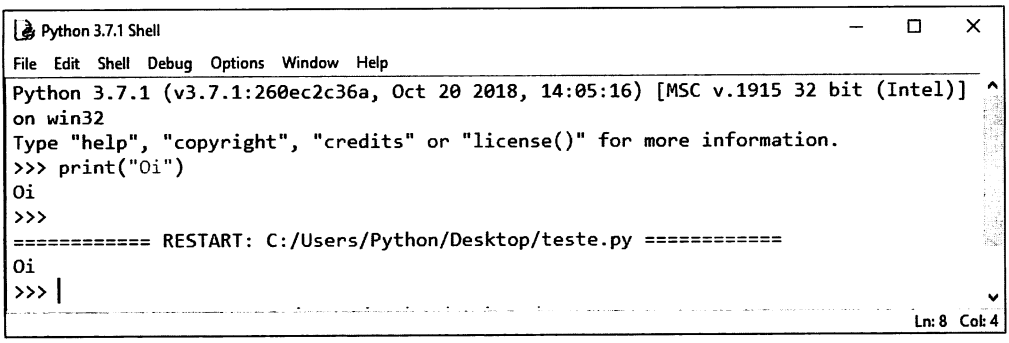


*Figura 2.20 – Janela do interpretador mostrando a execução do programa.*

Experimente modificar o texto entre aspas e executar novamente o programa, pressionando a tecla **F5** na janela do Editor ou selecionando **Run** no menu.

Agora estamos com tudo instalado para continuar aprendendo a programar. Durante todo o livro, programas de exemplo serão apresentados. Você deve digitá-los na janela do editor de textos, gravá-los e executá-los. Pode voltar a ler as seções 2.2 e 2.3 sempre que precisar. Com o tempo, essas tarefas serão memorizadas e realizadas sem esforço.

Experimente gravar seus programas em um diretório específico no computador. Isso ajudará a encontrá-los mais tarde. Uma dica é criar um diretório para cada capítulo do livro, mas você pode organizar os exemplos como quiser. Pode também fazer cópias dos exemplos e alterá-los sem receio. Embora seja possível baixar todos os exemplos já digitados, é altamente recomendado que você leia o livro e digite cada programa. Essa prática ajuda a memorizar as construções da linguagem e facilita a leitura dos programas.

## 2.4 Cuidados ao digitar seus programas

Ao digitar um programa no editor de textos, verifique se você o copiou exatamente como apresentado. Em Python, você deve tomar cuidado com os seguintes itens:

1. Letras maiúsculas e minúsculas são diferentes. Assim, **`print`** e **`Print`** são completamente diferentes, causando um erro caso você digite o P maiúsculo. No início, é comum termos o hábito de escrever a letra inicial de cada linha em letra maiúscula, causando erros em nossos programas. Em caso de erro, leia atentamente o que você digitou e compare com a listagem apresentada no livro.
2. Aspas são muito importantes e não devem ser esquecidas. Toda vez que abrir aspas, não se esqueça de fechá-las. Se você se esquecer, seu programa não funcionará. Observe que o IDLE muda a cor do texto entre aspas, facilitando essa verificação.
3. Parênteses não são opcionais em Python. Não remova os parênteses dos programas e preste a mesma atenção dada às aspas para abri-los e fechá-los. Todo parêntese aberto deve ser fechado.
4. Espaços são muito importantes. A linguagem Python se baseia na quantidade de espaço em branco antes do início de cada linha para realizar diversas operações, explicadas posteriormente no livro. Não se esqueça de digitar o texto dos programas com o mesmo alinhamento apresentado no livro. Observe também que o IDLE ajuda nesses casos, avisando sobre problemas de alinhamento. Nunca junte duas linhas em uma só até sentir-se seguro sobre como escrever corretamente em Python. Tente usar apenas espaços para alinhar seu programa, evite usar TABs (seu editor de textos pode converter TABs em espaços para você).

## 2.5 Primeiros programas

Vamos analisar nosso primeiro programa. A Figura 2.21 mostra a separação entre o nome da função, os parênteses, a mensagem e as aspas. É muito importante saber o nome de cada uma dessas partes para o correto entendimento dos conceitos e programas apresentados.

A função **`print`** informa que vamos exibir algo na tela. Pode-se dizer que a função exibe uma mensagem na tela do computador. Sempre que quisermos mostrar algo para o usuário do computador, como uma mensagem, uma pergunta ou o resultado de uma operação de cálculo, utilizaremos a função **`print`**.

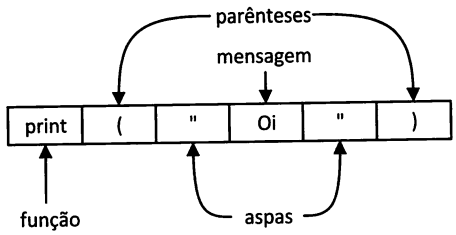


*Figura 2.21 – Nomes dos elementos do primeiro programa.*

Para separar os programas do texto explicativo do livro, todo trecho de programa, funções, variáveis e demais itens em Python serão apresentados com uma fonte de texto diferente (monoespaçada), como a utilizada para explicar a função **`print`**.

Voltando à Figura 2.21, temos as aspas como elementos importantes. Elas são utilizadas para separar textos destinados ao usuário do computador do resto do programa. Utilizamos aspas para indicar o início e o fim do texto de nossa mensagem. A mensagem é nosso texto em si e será exibida exatamente como digitada. Os parênteses são utilizados para separar os parâmetros de uma função, no caso, os de **`print`**. Um parâmetro é um valor passado para uma função: no caso da função **`print`**, a mensagem a imprimir (Oi).

O interpretador Python também pode ser utilizado como calculadora. Experimente digitar `2 + 3` no interpretador, como mostrado a seguir.

```
>>> 2 + 3
5
```

Não se esqueça de pressionar a tecla **Enter** para que o interpretador saiba que você terminou a digitação. O resultado deve ser apresentado na linha seguinte. Podemos também subtrair valores, como mostrado a seguir.

```
>>> 5 - 3
2
```

Podemos combinar adição e subtração na mesma linha:

```
>>> 10 - 4 + 2
8
```

A multiplicação é representada por um asterisco (*); e a divisão, pela barra (/):

```
>>> 2 * 10
20
>>> 20 / 4
5.0
```

A divisão nem sempre produz resultados inteiros. Por exemplo:

```
>>> 10 / 4
2.5
```

Se você deseja calcular a divisão inteira, ou seja, com resto, utilize duas barras em Python:

```
>>> 10 // 4
2
```

Falaremos mais sobre números fracionários e inteiros no Capítulo 3.

Para elevar um número a um expoente, utilizaremos dois asteriscos (**) para representar a operação de exponenciação (potenciação). Observe que não há qualquer espaço entre os dois asteriscos. Assim, para calcularmos $2^3$, escreveremos assim:

```
>>> 2 ** 3
8
```

**Lembrete**

A exponenciação é uma operação importante, vale lembrar que $2^3$ é igual a 2 x 2 x 2. No caso de $2^3$, 2 é a base e 3 o expoente, então lemos: "2 é elevado a potência de 3" ou simplesmente "2 elevado a 3".

A exponenciação tem também várias propriedades como:

Todo número elevado a zero é igual a 1: $X^0 = 1$

Todo número elevado a 1 é igual a si mesmo: $x^1 = x$

Um elevado a não importa qual número é 1: $1^x = 1$

Zero elevado a qualquer número positivo é igual a 0: $0^x = 0$

Podemos também calcular a raiz de um número pela exponenciação com expoente fracionário. Dessa forma, $a^{\frac{x}{y}}$ é igual a $\sqrt[y]{a^x}$, ou seja, $2^{\frac{1}{3}}$ é igual a $\sqrt[3]{2}$, e em Python:

```
>>> 2 ** 1/3
0.6666666666666666
```

Podemos também obter o resto da divisão de dois números usando o símbolo %. Assim, para calcularmos o resto da divisão entre 10 e 3, digitaríamos:

```
>>> 10 % 3
1
>>> 16 % 7
2
>>> 63 % 8
7
```

*Tabela 2.1 – Operadores e operações matemáticas*

| Operador | Operação |
|---|---|
| + | Adição |
| - | Subtração |
| * | Multiplicação |
| / | Divisão (com resultado fracionário) |
| // | Divisão (com resultados inteiros) |
| % | Módulo ou resto |
| ** | Exponenciação ou potenciação |

Os parênteses são utilizados em Python da mesma forma que em expressões matemáticas, ou seja, para alterar a ordem de execução de uma operação. Para relembrar a ordem de precedência das operações, temos as seguintes prioridades:

1. Exponenciação ou potenciação (**).
2. Multiplicação (*), divisão (/ e //) e módulo (%).
3. Adição (+) e subtração (-).

A expressão 1500 + (1500 * 5 / 100) é equivalente a:

$$1500 + \left(\frac{1500 \times 5}{100}\right)$$

Não se esqueça de que, tanto em Python como na matemática, as operações de mesma prioridade são realizadas da esquerda para a direita. Utilize parênteses sempre que precisar alterar a ordem de execução das operações e também para aumentar a clareza da fórmula.

**Exercício 2.1** Converta as seguintes expressões matemáticas para que possam ser calculadas usando o interpretador Python.

$10 + 20 \times 30$

$4^2 \div 30$

$(9^4 + 2) \times 6 - 1$

**Exercício 2.2** Digite a seguinte expressão no interpretador:

```
10 % 3 * 10 ** 2 + 1 - 10 * 4 / 2
```

Tente resolver o mesmo cálculo, usando apenas lápis e papel. Observe como a prioridade das operações é importante.

## 2.6 Conceitos de variáveis e atribuição

Além de operações simples de cálculo, o interpretador também pode ser usado para realizar operações mais complexas e mesmo executar programas completos. Antes de continuarmos, é importante observar o conceito de variáveis e como podemos usá-las em um programa. Em matemática, aprendemos o conceito de variável para representar incógnitas em equações do tipo `x + 1 = 2`, nas quais devemos determinar o valor de `x`, resolvendo a equação. Em programação, variáveis são utilizadas para armazenar valores e para dar nome a uma área de memória do computador onde armazenamos dados. Variáveis serão mais bem estudadas no Capítulo 3.

Por enquanto, podemos imaginar a memória do computador como uma grande estante, em que cada compartimento tem um nome. Para armazenar algo nesses compartimentos, usaremos o símbolo de igualdade (=) entre o nome do compartimento e o valor que queremos armazenar. Chamaremos essa operação de atribuição, na qual um valor é atribuído a uma variável. Quando lermos nosso programa, as operações de atribuição serão chamadas de "recebe", ou seja, uma variável recebe um valor.

A fim de simplificar as explicações de como um programa funciona, utilizaremos bolas pretas ❶ com números para relacionar uma determinada linha a um texto explicativo. Esses símbolos não fazem parte do programa e não devem ser digitados nem no interpretador nem no editor de textos.

Como quase tudo na vida, aprende-se a programar programando. Vamos escrever outro programa:

```
# Programa 2.1 - Primeiro programa com variáveis
a = 2 ❶
b = 3 ❷
print(a + b) ❸
```

Vejamos o que cada linha significa. A primeira linha inicia com #. O # é o símbolo usado para indicar que estamos comentando ou fazendo um comentário. Comentários são ignorados pelo interpretador Python e podemos escrever o que quisermos. Identificaremos os programas neste livro usando um comentário na primeira linha. Se você estiver digitando esse programa no interpretador, fique à vontade para ignorar os comentários, ou seja, não os digitar.

Em ❶, temos `a = 2`. Leia "a recebe 2". Essa linha diz que uma variável chamada `a` receberá o valor 2. Variáveis em programação têm o mesmo significado que em matemática. Você pode entender uma variável como uma forma de guardar

valores na memória do computador. Toda variável precisa ter um nome para que seu valor seja utilizado posteriormente. Esse conceito ficará mais claro um pouco mais adiante.

Em ❷, temos `b = 3`. Leia "b recebe 3". Essa linha realiza um trabalho muito parecido com o da linha anterior, mas a variável se chama `b`, e o valor é o número 3. Para entender o que faz essa linha, imagine que criamos um espaço na memória do computador para guardar outro valor, no caso, 3. Para podermos usar esse valor mais tarde, chamamos esse espaço de "b".

A linha ❸ solicita que o resultado da soma do conteúdo da variável `a` com o conteúdo da variável `b` seja exibido na tela. A função **`print`** realiza a impressão, mas, antes, o resultado de `a + b` é calculado. Veja que nessa linha estamos ordenando ao programa que calcule `a + b` e que exiba o resultado na tela. Como em matemática, passamos parâmetros ou valores para uma função usando parênteses. Esses parênteses são requeridos pelo interpretador Python. Vale se lembrar de *f(x)*; em que `f` é o nome da função, e `x` um parâmetro. No exemplo anterior, **`print`** é o nome da função; e o resultado de `a + b`, o valor passado como parâmetro. No decorrer deste livro, veremos diversas funções disponíveis no Python para realizar operações com o computador, como ler valores do teclado ou gravar dados em um arquivo.

Você pode experimentar o Programa 2.1 na janela do interpretador Python, como mostra a Seção 2.2. O resultado desse programa pode ser visto a seguir.

```
>>> a = 2
>>> b = 3
>>> print(a + b)
5
```

As duas primeiras linhas não enviam nada para a tela; por isso, apenas o resultado da terceira linha é mostrado.

Você pode estar se perguntando por que criamos duas variáveis, `a` e `b`, para somar dois números? Poderíamos ter obtido o mesmo resultado de diversas formas, como mostrado a seguir.

```
print(2 + 3)
```

Ou assim:

```
print(5)
```

Então, por que escolhemos resolver o problema usando variáveis? Primeiro, para podermos falar de variáveis, mas também para exemplificar uma grande diferença entre resolver um problema no papel e por meio de um computador.

Quando estamos resolvendo um problema de matemática no papel, como somar dois números, realizamos diversos cálculos mentalmente e escrevemos parte desse processo no papel, quando necessário. Depois de escrito no papel, mudar os valores não é tão simples.

Ao programarmos um computador, estamos transferindo esse cálculo para o computador. Como programar é descrever os passos para a solução do problema, é aconselhável escrevermos programas o mais claramente possível, de forma que possamos alterá-los caso precisemos e, mais importante, que possamos entendê-los mais tarde.

Quando escrevemos `print(2 + 3)`, o problema foi representado como sendo a soma de 2 e 3. Se precisarmos mudar as parcelas dessa soma, teremos de escrever outro programa. Isso também é válido para o primeiro programa, mas observe que, ao utilizarmos variáveis, estamos dando nome aos valores de entrada de nosso problema, aumentando, assim, a facilidade de entendermos o que o programa faz.

Já `print(5)` não descreve o problema em si. Estamos apenas ordenando ao computador que imprima o número 5 na tela. Não fizemos qualquer registro do que estávamos fazendo, ou de que nosso problema era somar dois números. Isso ficará mais claro no exemplo a seguir.

```
# Programa 2.2 - Cálculo de aumento de salário
salário = 1500 ❶
aumento = 5 ❷
print(salário + (salário * aumento / 100)) ❸
```

Em ❶ temos uma variável que é chamada `salário`, recebendo o valor 1500. Em ❷, outra variável, `aumento`, recebe o valor 5. Finalmente, em ❸ descrevemos a fórmula que calculará o valor do novo salário depois de receber um aumento. Teríamos, então, um resultado como:

```
>>> salário = 1500
>>> aumento = 5
>>> print(salário + (salário * aumento / 100))
1575.0
```

O Programa 2.2 pode ser escrito de forma mais direta, utilizando outra fórmula sem variáveis:

```
# Programa 2.3 - Alternativa para o cálculo de aumento de salário
print(1500 + (1500 * 5 / 100))
```

O objetivo desse exemplo é apresentar a diferença entre descrevermos o problema de forma genérica, separando os valores de entrada do cálculo. O resultado é idêntico: a diferença está na clareza da representação de nosso problema. Se mudarmos o valor do salário, no Programa 2.2, obteremos o resultado correto na saída do programa, sem precisar nos preocuparmos novamente com a fórmula do cálculo. Observe também que, se fizermos a mesma coisa no Programa 2.3, teremos de mudar o valor de salário em duas posições diferentes da fórmula, aumentando nossas chances de nos esquecermos de uma delas e, consequentemente, de recebermos um resultado incorreto.

Ao utilizarmos variáveis, podemos referenciar o mesmo valor várias vezes, sem nos esquecer de que podemos utilizar nomes mais significativos que simples `x` ou `y` para aumentar a clareza do programa. Por exemplo, no Programa 2.2, registramos a fórmula para o cálculo do aumento especificando o nome de cada variável, facilitando a leitura e o entendimento.

Se você já utilizou uma planilha eletrônica, como Microsoft Excel ou OpenOffice Calc, o conceito de variável pode ser entendido como as células de uma planilha eletrônica. Você pode escrever as fórmulas de sua planilha sem utilizar outras células, mas teria de reescrevê-las toda vez que os valores mudassem. Assim como as células de uma planilha eletrônica, as variáveis de um programa podem ser utilizadas diversas vezes e em lugares diferentes.

**Exercício 2.3** Faça um programa que exiba seu nome na tela.

**Exercício 2.4** Escreva um programa que exiba o resultado de $2a \times 3b$, em que $a$ vale 3 e $b$ vale 5.

**Exercício 2.5** Escreva um programa que calcule a soma de três variáveis e imprima o resultado na tela.

**Exercício 2.6** Modifique o Programa 2.2, de forma que ele calcule um aumento de 15% para um salário de R$ 750.

# CAPÍTULO 3

# Variáveis e entrada de dados

O capítulo anterior apresentou o conceito de variáveis, mas há mais por descobrir. Já sabemos que variáveis têm nomes que permitem acessar os valores dessas variáveis em outras partes do programa. Neste capítulo, vamos ampliar nosso conhecimento sobre variáveis, estudando novas operações e novos tipos de dados.

## 3.1 Nomes de variáveis

Em Python, nomes de variáveis devem iniciar obrigatoriamente com uma letra, mas podem conter números e o símbolo sublinha (_). Vejamos exemplos de nomes válidos e inválidos em Python na Tabela 3.1.

*Tabela 3.1 – Exemplo de nomes válidos e inválidos para variáveis*

| Nome | Válido | Comentários |
|---|---|---|
| `a1` | Sim | Embora contenha um número, o nome `a1` inicia com letra. |
| `velocidade` | Sim | Nome formado por letras. |
| `velocidade90` | Sim | Nome formado por letras e números, mas iniciado por letra. |
| `salário_médio` | Sim | O símbolo sublinha (_) é permitido e facilita a leitura de nomes grandes. |
| `salário médio` | Não | Nomes de variáveis não podem conter espaços em branco. |
| `_b` | Sim | O sublinha (_) é aceito em nomes de variáveis, mesmo no início. |
| `1a` | Não | Nomes de variáveis não podem começar com números. |

A versão 3 da linguagem Python permite a utilização de acentos em nomes de variáveis, pois, por padrão, os programas são interpretados utilizando-se um conjunto de caracteres chamado UTF-8 (`http://pt.wikipedia.org/wiki/Utf-8`), capaz de representar praticamente todas as letras dos alfabetos conhecidos.

Variáveis têm outras propriedades além de nome e conteúdo. Uma delas é conhecida como tipo e define a natureza dos dados que a variável armazena. Python tem vários tipos de dados, mas os mais comuns são números inteiros, números de ponto flutuante e strings. Além de poder armazenar números e letras, as variáveis em Python também armazenam valores como verdadeiro ou falso. Dizemos que essas variáveis são do tipo lógico. Veremos mais sobre variáveis do tipo lógico na Seção 3.3.

**TRÍVIA**

A maior parte das linguagens de programação foi desenvolvida nos Estados Unidos, ou considerando apenas nomes escritos na língua inglesa como nomes válidos. Por isso, acentos e letras consideradas especiais não são aceitos como nomes válidos na maior parte das linguagens de programação. Essa restrição é um problema não só para falantes do português, mas de muitas outras línguas que utilizam acentos ou mesmo outros alfabetos. O padrão americano é baseado no conjunto de caracteres ASCII (*), desenvolvido na década de 1960. Com a globalização da economia e o advento da internet, as aplicações mais modernas são escritas para trabalhar com um conjunto de caracteres dito universal, chamado Unicode (**).

(*) http://pt.wikipedia.org/wiki/Ascii
(**) http://pt.wikipedia.org/wiki/Unicode

## 3.2 Variáveis numéricas

Dizemos que uma variável é numérica quando armazena números inteiros ou de ponto flutuante.

Os números inteiros são aqueles sem parte decimal, como 1, 0, -5, 550, -47, 30000.

Números de ponto flutuante ou decimais são aqueles com parte decimal, como 1.0, 5.478, 10.478, 30000.4. Observe que 1.0, mesmo tendo zero na parte decimal, é um número de ponto flutuante.

Em Python, e na maioria das linguagens de programação, utilizamos o ponto, e não a vírgula, como separador entre a parte inteira e fracionária de um número. Essa é outra herança da língua inglesa. Observe também que não utilizamos nada como separador de milhar. Exemplo: 1.000.000 (um milhão) é escrito 1000000. Uma forma alternativa de separar números grandes é utilizar o sublinhado entre os dígitos (a partir do Python 3.6).

```
>>> 1_000  # Mil
1000
```

```
>>> 1_000_000  # Um milhão
1000000
>>> 1_0_0  # Cem, válido, mas a evitar
100
>>> 1_980.10  # _ pode ser combinado com ponto
1980.1
```

**Exercício 3.1** Complete a tabela a seguir, marcando inteiro ou ponto flutuante dependendo do número apresentado.

| Número | Tipo numérico |
|---|---|
| 5 | O inteiro O ponto flutuante |
| 5.0 | O inteiro O ponto flutuante |
| 4.3 | O inteiro O ponto flutuante |
| -2 | O inteiro O ponto flutuante |
| 100 | O inteiro O ponto flutuante |
| 1.333 | O inteiro O ponto flutuante |

### 3.2.1 Representação de valores numéricos

Internamente, todos os números são representados utilizando o sistema binário, ou seja, de base 2. Esse sistema permite apenas os dígitos 0 e 1. Para representar números maiores, combinamos vários dígitos, exatamente como fazemos com o sistema decimal, ou de base 10, que utilizamos.

Vejamos primeiro como isso funciona na base 10:

$$\begin{aligned} 531 &= 5 \times 10^2 + 3 \times 10^1 + 1 \times 10^0 \\ &= 5 \times 100 + 3 \times 10 + 1 \times 1 \\ &= 500 + 30 + 1 \\ &= 531 \end{aligned}$$

Multiplicamos cada dígito pela base elevada a um expoente igual ao número de casas à direita do dígito em questão. Como em 531 o 5 tem 2 dígitos à direita, multiplicamos $5 \times 10^2$. Para o 3, temos apenas outro dígito à direita, logo, $3 \times 10^1$. Finalmente, para o 1, sem dígitos à direita, temos $1 \times 10^0$. Somando esses componentes, temos o número 531. Fazemos isso tão rápido que é natural ou automático pensarmos desse jeito.

A mudança para o sistema binário segue o mesmo processo, mas a base agora é 2, e não 10. Assim, 1010 em binário representa:

$$1010 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$
$$= 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$$
$$= 8 + 0 + 2 + 0$$
$$= 10$$

A utilização do sistema binário é transparente em Python, ou seja, se você não solicitar explicitamente que esse sistema seja usado, tudo será apresentado na base 10 utilizada no dia a dia. A importância da noção de diferença de base é importante, pois ela explica os limites da representação. O limite de representação é o valor mínimo e máximo que pode ser representado em uma variável numérica. Esse limite é causado pela quantidade de dígitos que foram reservados para armazenar o número em questão. Vejamos como funciona na base 10.

Se você tem apenas 5 dígitos para representar um número, o maior número é 99999. E o menor seria (-99999). O mesmo acontece no sistema binário, sendo que lá reservamos um dígito para registrar os sinais de positivo e negativo.

Para números inteiros, Python utiliza um sistema de precisão ilimitada que permite a representação de números muito grandes. É como se você sempre pudesse escrever novos dígitos à medida que for necessário. Você pode calcular em Python valores como $2^{1000000}$ (`2 ** 1000000`) sem problemas de representação, mesmo quando o resultado é um número de 301030 dígitos.

Em ponto flutuante, temos limite e problemas de representação. Um número decimal é representado em ponto flutuante utilizando-se uma mantissa e um expoente (*sinal* × *mantissa* × $base^{expoente}$). Tanto a mantissa quanto o expoente têm um número de dígitos máximos que limita os números que podem ser representados. Você não precisa se preocupar com isso no momento, pois esses valores são bem grandes e você não terá problemas na maioria de seus programas. Obtenha mais informações acessando `http://pt.wikipedia.org/wiki/Ponto_flutuante`.

A versão 3.7 do Python tem como limites $2.2250738585072014 \times 10^{-308}$ e $1.7976931348623157 \times 10^{308}$, suficientemente grandes e pequenos para quase qualquer tipo de aplicação. É possível encontrar problemas de representação em função de como os números decimais são convertidos em números de ponto flutuante. Esses problemas são bem conhecidos e afetam todas as linguagens de programação, não sendo um problema específico do Python.

Vejamos um exemplo: o número 0.1 não tem nada de especial no sistema decimal, mas é uma dízima periódica no sistema binário. Você não precisa se preocupar com esses detalhes agora, mas pode investigá-los mais tarde quando precisar

(normalmente cursos de computação apresentam uma disciplina, chamada cálculo numérico, para abordar esses tipos de problemas). Digite no interpretador

```
3 * 0.1
```

Você deve ter obtido como resultado 0.30000000000000004, e não 0.3, como esperado. Não se assuste: não é um problema de seu computador, mas de representação. Se for necessário, durante seus estudos, cálculos mais precisos, ou se os resultados em ponto flutuante não satisfizerem os requisitos de precisão esperados, verifique os módulos `decimals` e `fractions`. A documentação do Python traz uma página específica sobre esse tipo de problema: `http://docs.python.org/py3k/tutorial/floatingpoint.html`.

Você também pode trabalhar com a base 2 (binária), base 8 (octal) ou base 16 (hexadecimal) em Python. Embora seja um recurso pouco utilizado, pode ser interessante para você no futuro. É possível entrar números em binário utilizando o prefixo `0b` (zero b), em octal com o prefixo 0o (zero ó) e em hexadecimal 0x (zero x):

```
>>> a = 0b10  # Base 2 - binário
>>> a
2
>>> b = 0x10  # Base 16 - hexadecimal
>>> b
16
>>> c = 0o10  # Base 8 - octal
>>> c
8
```

Observe que, independentemente da base da dados utilizada para introduzir o número, este é mostrado por padrão na base 10. A base que você utilizou inicialmente para representar o número não é armazenada na variável em si, apenas o valor já convertido.

## 3.3 Variáveis do tipo Lógico

Muitas vezes, queremos armazenar um conteúdo simples: verdadeiro ou falso em uma variável. Nesse caso, utilizaremos um tipo de variável chamado tipo lógico ou booleano. Em Python, escreveremos **`True`** para verdadeiro e **`False`** para falso. Observe que o T e o F são escritos com letras maiúsculas:

```
resultado = True
aprovado = False
```

### 3.3.1 Operadores relacionais

Para realizarmos comparações lógicas, utilizaremos operadores relacionais. A lista de operadores relacionais suportados em Python é apresentada na Tabela 3.2.

*Tabela 3.2 – Operadores relacionais*

| Operador | Operação | Símbolo matemático |
|---|---|---|
| == | igualdade | = |
| > | maior que | > |
| < | menor que | < |
| != | diferente | ≠ |
| >= | maior ou igual | ≥ |
| <= | menor ou igual | ≤ |

O resultado de uma comparação com esses operadores é um valor do tipo lógico, ou seja, **True** (verdadeiro) ou **False** (falso). Utilizaremos o verbo "avaliar" para indicar a resolução de uma expressão. Por exemplo:

```
>>> a = 1          # a recebe 1
>>> b = 5          # b recebe 5
>>> c = 2          # c recebe 2
>>> d = 1          # d recebe 1
>>> a == b         # a é igual a b ?
False
>>> b > a          # b é maior que a?
True
>>> a < b          # a é menor que b?
True
>>> a == d         # a é igual a d?
True
>>> b >= a         # b é maior ou igual a a?
True
>>> c <= b         # c é menor ou igual a b?
True
>>> d != a         # d é diferente de a?
False
>>> d != b         # d é diferente de b?
True
```

Esses operadores são utilizados como na matemática. Especial atenção deve ser dada aos operadores >= e <=. O resultado desses operadores é realmente maior ou igual e menor ou igual, ou seja, `5 >= 5` é verdadeiro, assim como `5 <= 5`.

Observe que utilizamos o símbolo de cerquilha (#) para escrever comentários na linha de comando. Todo texto à direita do cerquilha é ignorado pelo interpretador Python, ou seja, você pode escrever o que quiser. Veja como é mais fácil entender cada linha quando a comentamos. Comentários não são obrigatórios, mas são muito importantes. Você pode e deve utilizá-los em seus programas para facilitar o entendimento e oferecer uma anotação para si mesmo.

É comum lermos um programa alguns meses depois de escrito e termos dificuldade de lembrar o que realmente queríamos fazer. Você também não precisa comentar todas as linhas de seus programas ou escrever o óbvio. Uma dica é identificar os programas com seu nome, a data em que começou a ser escrito e mesmo a listagem ou capítulo do livro onde você o encontrou.

Variáveis de tipo lógico também podem ser utilizadas para armazenar o resultado de expressões e comparações:

```
nota = 8
média = 7
aprovado = nota > média
print(aprovado)
```

Se uma expressão contém operações aritméticas, estas devem ser calculadas antes que os operadores relacionais sejam avaliados. Quando avaliamos uma expressão, substituímos o nome das variáveis por seu conteúdo e só então verificamos o resultado da comparação.

**Exercício 3.2** Complete a tabela a seguir, respondendo True ou False. Considere `a = 4, b = 10, c = 5.0, d = 1` e `f = 5`.

| Expressão | Resultado |
|---|---|
| `a == c` | ○ True ○ False |
| `a < b` | ○ True ○ False |
| `d > b` | ○ True ○ False |
| `c != f` | ○ True ○ False |
| `a == b` | ○ True ○ False |
| `c < d` | ○ True ○ False |

| Expressão | Resultado |
|---|---|
| `b > a` | ○ True ○ False |
| `c >= f` | ○ True ○ False |
| `f >= c` | ○ True ○ False |
| `c <= c` | ○ True ○ False |
| `c <= f` | ○ True ○ False |
| | |

### 3.3.2 Operadores lógicos

Para agrupar operações com lógica booleana, utilizaremos operadores lógicos. Python suporta três operadores básicos: **not** (não), **and** (e), **or** (ou). Esses operadores podem ser traduzidos como não (¬ negação), e (Λ conjunção) e ou (V disjunção).

*Tabela 3.3 – Operadores lógicos*

| Operador Python | Operação |
|---|---|
| **not** | não |
| **and** | e |
| **or** | ou |

Cada operador obedece a um conjunto simples de regras, expresso pela tabela verdade desse operador. A tabela verdade demonstra o resultado de uma operação com um ou dois valores lógicos ou operandos. Quando o operador utiliza apenas um operando, dizemos que é um operador unário. Ao utilizar dois operandos, é chamado operador binário. O operador de negação (**not**) é um operador unário. **or** (ou) e **and** (e) são operadores binários, precisando, assim, de dois operandos.

#### 3.3.2.1 Operador not

O operador **not** (não) é o mais simples, pois precisa apenas de um operador. A operação de negação também é chamada de inversão, pois um valor verdadeiro negado se torna falso e vice-versa. A tabela verdade do operador **not** (não) é apresentada na Tabela 3.4.

*Tabela 3.4 – Tabela verdade do operador not (não)*

| $V_1$ | not $V_1$ |
|---|---|
| V | F |
| F | V |

Exemplo:

```
>>> not True
False
>>> not False
True
```

#### 3.3.2.2 Operador and

O operador **and** (e) tem sua tabela verdade representada na Tabela 3.5. O operador **and** (e) resulta verdadeiro apenas quando seus dois operadores forem verdadeiros.

*Tabela 3.5 – Tabela verdade do operador and (e)*

| $V_1$ | $V_2$ | $V_1$ and $V_2$ |
|---|---|---|
| V | V | V |
| V | F | F |
| F | V | F |
| F | F | F |

Exemplos:

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

#### 3.3.2.3 Operador or

A tabela verdade do operador **or** (ou) é apresentada na Tabela 3.6. A regra fundamental do operador **or** (ou) é que ele resulta em falso apenas se seus dois operadores também forem falsos. Se apenas um de seus operadores for verdadeiro, ou se os dois forem, o resultado da operação será verdadeiro.

*Tabela 3.6 – Tabela verdade do operador or(ou)*

| $V_1$ | $V_2$ | $V_1$ or $V_2$ |
|---|---|---|
| V | V | V |
| V | F | V |
| F | V | V |
| F | F | F |

Exemplos:

```
>>> True or True
True
>>> True or False
True
```

```
>>> False or True
True
>>> False or False
False
```

**Exercício 3.3** Complete a tabela a seguir utilizando `a = True`, `b = False` e `c = True`.

| Expressão | Resultado |
|---|---|
| `a and a` | O True O False |
| `b and b` | O True O False |
| `not c` | O True O False |
| `not b` | O True O False |
| `not a` | O True O False |
| `a and b` | O True O False |
| `b and c` | O True O False |

| Expressão | Resultado |
|---|---|
| `a or c` | O True O False |
| `b or c` | O True O False |
| `c or a` | O True O False |
| `c or b` | O True O False |
| `c or c` | O True O False |
| `b or b` | O True O False |
| | |

### 3.3.3 Expressões lógicas

Os operadores lógicos podem ser combinados em expressões lógicas mais complexas. Quando uma expressão tiver mais de um operador lógico, avalia-se o operador **not** (não) primeiro, seguido do operador **and** (e) e, finalmente, **or** (ou). Vejamos a seguir a ordem de avaliação da expressão, onde a operação sendo avaliada é sublinhada; e o resultado, mostrado na linha seguinte.

```
True or False and not True
True or False and False
True or False
True
```

Os operadores relacionais também podem ser utilizados em expressões com operadores lógicos.

```
salário > 1000 and idade > 18
```

Nesses casos, os operadores relacionais devem ser avaliados primeiro. Façamos `salário = 100` e `idade = 20`. Teremos:

```
salário > 1000 and idade > 18
100 > 1000 and 20 > 18
False and True
False
```

A grande vantagem de escrever esse tipo de expressão é representar condições que podem ser avaliadas com valores diferentes. Por exemplo: imagine que `salário > 1000 and idade > 18` seja uma condição para um empréstimo de compra de um carro novo. Quando `salário = 100` e `idade = 20`, sabemos que o resultado da expressão é falso, e podemos interpretar que, nesse caso, a pessoa não receberia o empréstimo. Avaliemos a mesma expressão com `salário = 2000` e `idade = 30`.

```
salário > 1000 and idade > 18
2000 > 1000 and 30 > 18
True and True
True
```

Agora o resultado é **`True`** (verdadeiro) e poderíamos dizer que a pessoa atende às condições para obter o empréstimo.

**Exercício 3.4** Escreva uma expressão para determinar se uma pessoa deve ou não pagar imposto. Considere que pagam imposto pessoas cujo salário é maior que R$ 1.200,00.

**Exercício 3.5** Calcule o resultado da expressão `A > B and C or D`, utilizando os valores da tabela a seguir.

| A | B | C | D | Resultado |
|---|---|---|---|---|
| 1 | 2 | True | False | |
| 10 | 3 | False | False | |
| 5 | 1 | True | True | |

**Exercício 3.6** Escreva uma expressão que será utilizada para decidir se um aluno foi ou não aprovado. Para ser aprovado, todas as médias do aluno devem ser maiores que 7. Considere que o aluno cursa apenas três matérias, e que a nota de cada uma está armazenada nas seguintes variáveis: `matéria1`, `matéria2` e `matéria3`.

## 3.4 Variáveis string

Variáveis do tipo string armazenam cadeias de caracteres como nomes e textos em geral. Chamamos cadeia de caracteres uma sequência de símbolos como letras, números, sinais de pontuação etc. Exemplo: João e Maria comem pão. Nesse caso, João é uma sequência com as letras J, o, ã, o. Para simplificar o texto, utilizaremos o nome string para mencionar cadeias de caracteres. Podemos imaginar uma string como uma sequência de blocos, em que cada letra, número ou espaço em branco ocupa uma posição, como mostra a Figura 3.1.

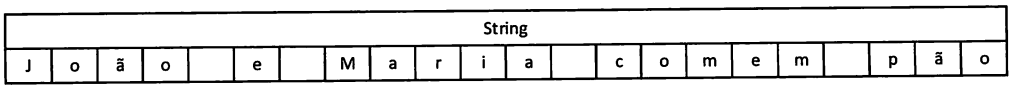


*Figura 3.1 – Representação de uma string.*

Para possibilitar a separação entre o texto do seu programa e o conteúdo de uma string, utilizaremos aspas (") para delimitar o início e o fim da sequência de caracteres.

Voltando ao exemplo anterior, escreveremos `"João e Maria comem pão"`. Veja que nesse caso não há qualquer problema em utilizarmos espaços. Na verdade, o computador ignora praticamente tudo que escrevemos entre aspas, mas veremos mais tarde que não é bem assim.

As variáveis do tipo string são utilizadas para armazenar sequências de caracteres, normalmente utilizadas em textos ou mensagens. O tipo string é muito útil e bastante utilizado para exibir mensagens ou mesmo para gerar outros arquivos.

Uma string em Python tem um tamanho associado, assim como um conteúdo que pode ser acessado caractere a caractere. O tamanho de uma string pode ser obtido utilizando-se a função **`len`**. Essa função retorna o número de caracteres na string. Dizemos que uma função retorna um valor quando podemos substituir o texto da função por seu resultado. A função **`len`** retorna um valor do tipo inteiro, representando a quantidade de caracteres contidos na string. Se a string é vazia (representada simplesmente por `""`, ou seja, duas aspas sem nada entre elas, nem mesmo espaços em branco), seu tamanho é igual a zero. Façamos alguns testes:

```
>>> len("A")
1
>>> len("AB")
2
>>> len("")
0
>>> len("O rato roeu a roupa")
19
```

Como dito anteriormente, outra característica de strings é poder acessar seu conteúdo caractere a caractere. Sabendo que uma string tem um determinado tamanho, podemos acessar seus caracteres utilizando um número inteiro para representar sua posição. Esse número é chamado de índice, e começamos a contar de zero. Isso quer dizer que o primeiro caractere da string é de posição ou índice 0. Observe a Figura 3.2.

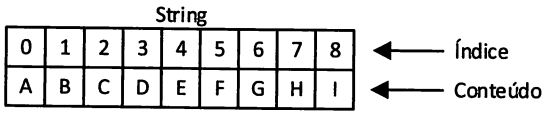


*Figura 3.2 – Índices e conteúdo de uma variável string.*

Para acessar os caracteres de uma string, devemos informar o índice ou posição do caractere entre colchetes ([ ]). Como o primeiro caractere de uma string é o de índice 0, podemos acessar valores de 0 até o tamanho da string menos 1. Logo, se a string contiver 9 caracteres, poderemos acessar os caracteres de 0 a 8. Se tentarmos acessar um índice maior que a quantidade de caracteres da string, o interpretador emitirá uma mensagem de erro. Veja o resultado de alguns testes com strings:

```
>>> a = "ABCDEF"
>>> a[0]
A
>>> a[1]
B
>>> a[5]
F
>>> a[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> len(a)
6
```

### 3.4.1 Operações com strings

As variáveis de tipo string suportam operações como fatiamento, concatenação e composição. Por fatiamento, podemos entender a capacidade de utilizar apenas uma parte de uma string, ou uma fatia. A concatenação nada mais é que poder juntar duas ou mais strings em uma nova string maior. A composição é muito utilizada em mensagens que enviamos à tela e consiste em utilizar strings como modelos em que podemos inserir outras strings. Veremos cada uma dessas operações nas seções a seguir.

### 3.4.1.1 Concatenação

O conteúdo de variáveis string podem ser somados, ou melhor, concatenados. Para concatenar duas strings, utilizamos o operador de adição (+). Assim, `"AB" + "C"` é igual a `"ABC"`. Um caso especial de concatenação é a repetição de uma string várias vezes. Para isso, utilizamos o operador de multiplicação (*): `"A" * 3` é igual a `"AAA"`. Vejamos alguns exemplos:

```
>>> s = "ABC"
>>> s + "C"
ABCC
>>> s + "D" * 4
ABCDDDD
>>> "X" + "-" * 10 + "X"
X----------
>>> s + "x4 = " + s * 4
ABCx4 = ABCABCABCABC
```

NOTA: esta forma de concatenação só pode ser usada com strings.

```
>>> nome = "Tião"
>>> idade = 10
>>> nome + idade
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    nome + idade
TypeError: can only concatenate str (not "int") to str
```

Se você precisar combinar variáveis de tipos diferentes para formar uma string maior, use composição apresentada na seção seguinte.

### 3.4.1.2 Composição

Juntar várias strings para construir uma mensagem nem sempre é prático. Por exemplo, exibir que `"João tem X anos"`, em que X é uma variável numérica.

Usando a composição de strings do Python, podemos escrever de forma simples e clara:

```
"João tem %d anos" % X
```

Em que o símbolo de `%` foi utilizado para indicar a composição da string anterior com o conteúdo da variável `X`. O `%d` dentro da primeira string é o que chamamos de marcador de posição. O marcador indica que naquela posição estaremos colocando um valor inteiro, daí o `%d`.

Python suporta diversas operações com marcadores. Veremos mais sobre marcadores em outras partes do livro. A Tabela 3.7 apresenta os principais tipos de marcadores. Veja que eles são diferentes, de acordo com o tipo de variável que vamos utilizar.

*Tabela 3.7 – Marcadores*

| Marcador | Tipo |
|---|---|
| %d | Números inteiros |
| %s | Strings |
| %f | Números decimais |

Imagine que precisamos apresentar um número como 001 ou 002, mas que também pode ser algo como 050 ou 561. Nesse caso, estamos querendo apresentar um número com três posições, completando com zeros à esquerda se o número for menor. Podemos realizar essa operação utilizando `"%03d" % X`. Observe que adicionamos 03 entre o `%` e o `d`. Se você precisar apenas que o número ocupe três posições, mas não desejar zeros à esquerda, basta retirar o zero e utilizar `"%3d" % X`. Isso é muito importante quando estamos gravando dados em um arquivo ou simplesmente exibindo informações na tela. Vejamos alguns exemplos:

```
>>> idade = 22
>>> "[%d]" % idade
[22]
>>> "[%03d]" % idade
[022]
>>> "[%3d]" % idade
[ 22]
>>> "[%-3d]" % idade
[22 ]
```

Quando formatamos números decimais, podemos utilizar dois valores entre o símbolo de `%` e a letra `f`. O primeiro indica o tamanho total em caracteres a reservar; e o segundo, o número de casas decimais. Assim, `%5.2f` diz que estaremos imprimindo um número decimal utilizando cinco posições, sendo que duas são para a parte decimal. Isso é muito interessante para exibir o resultado de cálculos ou representar dinheiro. Por exemplo, para exibir R$ 5, você pode utilizar `"R$%f" % 5`, mas o resultado não é bem o que esperamos, pois normalmente utilizamos apenas dois dígitos após a vírgula quando falamos de dinheiro. Vejamos alguns exemplos:

```
>>> "%5f" % 5
5.000000
```

```
>>> "%5.2f" % 5
 5.00
>>> "%10.5f" % 5
   5.00000
```

O poder da composição realmente aparece quando precisamos combinar vários valores em uma nova string. Imagine que João tem 22 anos e apenas R$ 51,34 no bolso. Para exibir essa mensagem, podemos utilizar:

```
"%s tem %d anos e apenas R$%5.2f no bolso" % ("João", 22, 51.34)
```

Python suporta diversas operações com marcadores. Veremos mais sobre marcadores em outras partes do livro. Quando temos mais de um marcador na string, somos obrigados a escrever os valores a substituir entre parênteses. Agora, vejamos exemplos com outros tipos, e utilizando mais de uma variável na composição:

```
>>> nome = "João"
>>> idade = 22
>>> grana = 51.34
>>> "%s tem %d anos e R$%f no bolso." % (nome, idade, grana)
João tem 22 anos e R$51.340000 no bolso.
>>> "%12s tem %3d anos e R$%5.2f no bolso." % (nome, idade, grana)
        João tem  22 anos e R$51.34 no bolso.
>>> "%12s tem %03d anos e R$%5.2f no bolso." % (nome, idade, grana)
        João tem 022 anos e R$51.34 no bolso.
>>> "%-12s tem %-3d anos e R$%-5.2f no bolso." % (nome, idade, grana)
João         tem 22  anos e R$51.34 no bolso.
```

Essa forma de composição foi substituída nas versões mais modernas do interpretador, e, embora continue a ser válida, tem caído em desuso.

Uma forma mais moderna de compor strings é utilizando o método **format**. Vejamos o programa anterior, mas reescrito usando **format** em vez de %:

```
>>> nome = "João"
>>> idade = 22
>>> grana = 51.34
>>> "{} tem {} anos e R${} no bolso.".format(nome, idade, grana)
'João tem 22 anos e R$51.34 no bolso.'
>>> "{:12} tem {:3} anos e R${:5.2f} no bolso.".format(nome, idade, grana)
'João         tem  22 anos e R$51.34 no bolso.'
>>> "{:12} tem {:03} anos e R${:5.2f} no bolso.".format(nome, idade, grana)
'João         tem 022 anos e R$51.34 no bolso.'
>>> "{:<12s} tem {:<3} anos e R${:5.2f} no bolso.".format(nome, idade, grana)
'João         tem 22  anos e R$51.34 no bolso.'
```

Essa forma de usar a composição é mais compacta e mais poderosa que a antiga forma que utiliza o `%`. É interessante conhecer a formatação com `%` para ler programas mais antigos que ainda usam essa notação.

Basicamente, substituímos o `%` por `.format` e os `%d`, `%s`, `%f` por `{}`. Com `format`, você não é obrigado a especificar o tipo. Veja também que escrevemos o tamanho da máscara após os `:` por exemplo `{:12}`, enquanto na notação antiga utilizaríamos `%12d`. No caso da formatação de dinheiro, precisamos usar `{:5.2f}` para imprimir de forma equivalente a `%5.2f`. No caso das máscaras com sinal `-`, veja que as substituímos por `<`: `%-12s` passou a `{:<12}`.

Como Python é uma linguagem que não para de evoluir, uma terceira forma de compor strings foi adicionada na versão 3.6. Ela se chama f-string, pois escrevemos a letra f antes de abrirmos as aspas. Vejamos o programa anterior, mas dessa vez usando f-strings:

```
>>> nome = "João"
>>> idade = 22
>>> grana = 51.34
>>> f"{nome} tem {idade} anos e R${grana} no bolso."
'João tem 22 anos e R$51.34 no bolso.'
>>> f"{nome:12} tem {idade:3} anos e R${grana:5.2f} no bolso."
'João         tem  22 anos e R$51.34 no bolso.'
>>> f"{nome:12} tem {idade:03} anos e R${grana:5.2f} no bolso."
'João         tem 022 anos e R$51.34 no bolso.'
>>> f"{nome:<12s} tem {idade:<3} anos e R${grana:5.2f} no bolso."
'João         tem 22  anos e R$51.34 no bolso.'
```

Com f-strings, além de prefixarmos a string com a letra f, escrevemos o nome da variável diretamente na string, entre `{}`. Essa forma é mais compacta que utilizar o método `format` e é a que utilizaremos no restante do livro. As regras de formatação após os `:` são idênticas às descritas para o `format`.

Não se preocupe com esses detalhes por enquanto, pois, nos capítulos 5 e 7, veremos mais sobre composição e formatação de strings.

#### 3.4.1.3 Fatiamento de strings

O fatiamento em Python é muito poderoso. Imagine nossa string de exemplo da Figura 3.2. Podemos fatiá-la de forma a escrever apenas seus dois primeiros caracteres AB utilizando como índice `[0:2]`. O fatiamento funciona com a utilização de dois pontos no índice da string. O número à esquerda dos dois pontos indica a posição de início da fatia; e o à direita, do fim. No entanto, é preciso ter atenção ao final, pois no exemplo anterior utilizamos 2; e o C, que é o caractere na posição 2,

não foi incluído. Dizemos que isso acontece porque o final da fatia não é incluído nela, sendo deixado de fora. Entenda `[0:2]` como a fatia de caracteres da posição 0 até a posição 2, sem incluí-la, ou o intervalo fechado em 0 e aberto em 2.

Vejamos outros exemplos de fatias:

```
>>> s = "ABCDEFGHI"
>>> s[0:2]
AB
>>> s[1:2]
B
>>> s[2:4]
CD
>>> s[0:5]
ABCDE
>>> s[1:8]
BCDEFGH
```

Podemos também omitir o número da esquerda ou o da direita para representar do início ou até o final. Assim, `[:2]` indica do início até o segundo caractere (sem incluí-lo), e `[1:]` indica do caractere de posição 1 até o final da string. Observe que, nesse caso, nem precisamos saber quantos caracteres a string contém. Se omitirmos o início e o fim da fatia `[:]`, estaremos fazendo apenas uma cópia de todos os caracteres da string para uma nova string.

Podemos também utilizar valores negativos para indicar posições a partir da direita. Assim `-1` é o último caractere; `-2`, o penúltimo; e assim por diante. Veja o resultado de testes com índices negativos:

```
>>> s = "ABCDEFGHI"
>>> s[:2]
AB
>>> s[1:]
BCDEFGHI
>>> s[0:-2]
ABCDEFG
>>> s[:]
ABCDEFGHI
>>> s[-1:]
I
>>> s[-5:7]
EFG
>>> s[-2:-1]
H
```

Veremos mais sobre strings em Python no Capítulo 7.

## 3.5 Sequências e tempo

Um programa é executado linha por linha pelo computador, executando as operações descritas no programa uma após a outra. Quando trabalhamos com variáveis, devemos nos lembrar de que o conteúdo de uma variável pode mudar com o tempo. Isso porque, a cada vez que alteramos o valor de uma variável, o valor anterior é substituído pelo novo.

No programa seguinte, a variável `dívida` foi utilizada para registrar quanto alguém estava devendo; e a variável `compra`, o valor de novas despesas dessa pessoa. Como somos justos, a pessoa começou sem dívidas em ❶.

```
# Programa 3.1 - Exemplo de sequência e tempo
dívida = 0 ❶
compra = 100 ❷
dívida = dívida + compra ❸
compra = 200 ❹
dívida = dívida + compra ❺
compra = 300 ❻
dívida = dívida + compra ❼
compra = 0 ❽
print(dívida) ❾
```

Em ❷, temos a primeira compra no valor de R$ 100. No entanto, o valor da dívida continua sendo 0, pois ainda não alteramos seu valor de forma a adicionar a compra realizada. Isso é feito em ❸. Observe que estamos atualizando o valor da dívida com o valor atual mais a compra.

Em ❷, registramos uma nova compra no valor de R$ 200. Nesse ponto, a compra tem seu valor substituído por R$ 200, causando a perda do valor anterior, R$ 100. Como já somamos o valor anterior na variável dívida, essa perda não representará um problema.

❺ é idêntica a ❸, mas o resultado é bem diferente. Nesse momento, a compra vale R$ 200; e a dívida, R$ 100.

Em ❻, alteramos o valor de compra novamente. Dessa vez, a compra foi de R$ 300.

❼ é idêntica a ❸ e ❺, mas seu resultado é diferente, pois nesse momento temos a compra valendo R$ 300; e a dívida igual a R$ 300 (100 + 200), sendo atualizada para R$ 600 (300 + 300).

Em ❽, simplesmente dizemos que a compra foi 0, representando que a pessoa não comprou mais nada.

❾ exibe o conteúdo da variável dívida na tela (600).

A Figura 3.3 mostra a evolução do conteúdo de nossas duas variáveis em função do tempo, representado pelo número da linha.

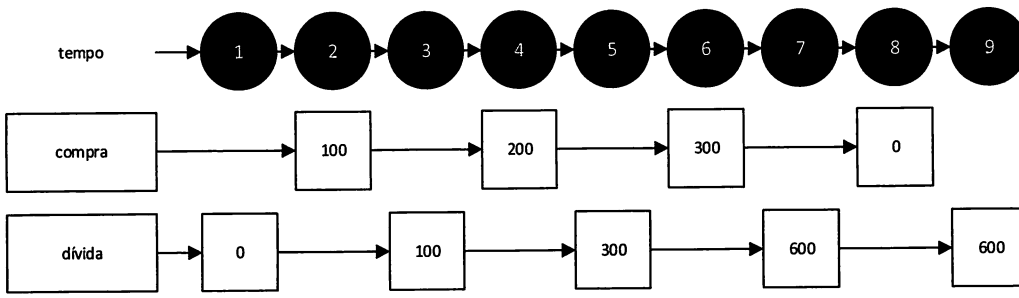


*Figura 3.3 – Mudanças no valor de duas variáveis no tempo.*

## 3.6 Rastreamento

Uma das principais diferenças entre ler um texto e um programa é justamente seguir as mudanças de valores de cada variável conforme o programa é executado. Entender que o valor das variáveis pode mudar durante a execução do programa não é tão natural, mas é fundamental para a programação de computadores. Um programa não pode ser lido como um simples texto, mas cuidadosamente analisado linha a linha. Ao escrever seus programas, verifique linha a linha os efeitos e mudanças causados no valor de cada variável.

Para programar corretamente, você deve ser capaz de entender o que cada linha do programa significa e os efeitos que produz. Essa atividade, chamada de rastreamento, é muito importante para entender novos programas e para encontrar erros nos programas que você escreverá.

Para rastrear um programa, utilize lápis, borracha e uma folha de papel. Escreva o nome de suas variáveis em uma folha de papel, como se fossem títulos de colunas, deixando espaço para ser preenchido embaixo desses nomes. Leia uma linha do programa de cada vez e escreva o valor atribuído a cada variável na outra folha, na mesma coluna em que escreveu o nome da variável.

Se o valor da variável mudar, escreva o novo valor e risque o anterior, um embaixo do outro, formando uma coluna. Ao exibir algo na tela, escreva também na outra folha, como se ela fosse a sua tela (você pode desenhar a tela como se fosse uma variável, mas lembre-se de deixar um pouco mais de espaço). Um exemplo de como ficaria o resultado do rastreamento do Programa 3.1 é apresentado na Figura 3.4.

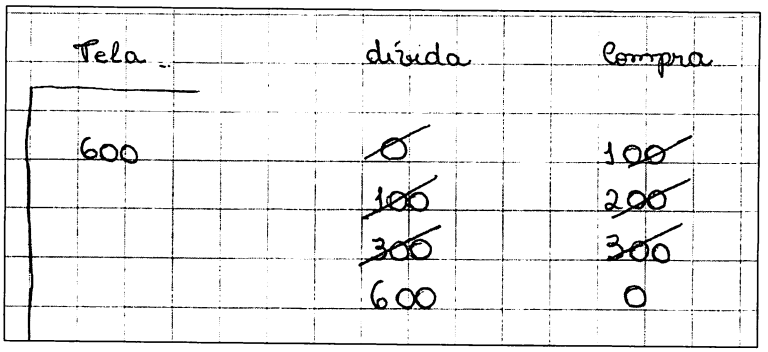


*Figura 3.4 – Exemplo de rastreamento no papel.*

O rastreamento vai ajudá-lo a entender melhor as mudanças de valores de suas variáveis e a acompanhar a execução do programa, como mais tarde será feito pelo computador. É um processo detalhado que precisa de atenção. Não tente simplificá-lo ou começar a rastrear no meio de um programa. Você deve rastrear linha a linha, do início ao fim do programa. Se encontrar um erro, pode parar o rastreamento e corrigi-lo, mas lembre-se de recomeçar do início sempre que alterar o programa ou os valores sendo rastreados.

Dominar o rastreamento de um programa é essencial para programar e ajuda muito a entender como os programas realmente funcionam. Lembre-se de que programar é detalhar, e que simplesmente ler o texto de um programa não é suficiente. Você deve rastreá-lo para entendê-lo. Embora pareça óbvio, esse é um dos erros mais comuns quando se começa a programar. Se um programa não funciona ou se você não entendeu exatamente o que ele faz, o rastreamento é a melhor ferramenta para descobrir o que está acontecendo.

## 3.7 Entrada de dados

Até agora, nossos programas trabalharam apenas com valores conhecidos, escritos no próprio programa. No entanto, o melhor da programação é poder escrever a solução de um problema e aplicá-la várias vezes. Para isso, precisamos melhorar nossos programas de forma a permitir que novos valores sejam fornecidos durante sua execução, de modo que poderemos executá-los com valores diferentes sem alterar os programas em si.

Chamamos de entrada de dados o momento em que seu programa recebe dados ou valores por um dispositivo de entrada de dados (como o teclado do computador) ou de um arquivo em disco.

A função **input** é utilizada para solicitar dados do usuário. Ela recebe um parâmetro, que é a mensagem a ser exibida, e retorna o valor digitado pelo usuário. Vejamos um exemplo:

```
x = input("Digite um número: ")
print(x)
```

Que produz a seguinte saída na tela (se você digitar o 5 e pressionar a tecla **ENTER**):

```
Digite um número: 5
5
```

Vejamos outro exemplo:

```
nome = input("Digite seu nome:") ❶
print(f"Você digitou {nome}")
print(f"Olá, {nome}!")
```

Em ❶, solicitamos a entrada de dados, no caso, o nome do usuário. A mensagem "Digite seu nome:" é exibida, e o programa para até que o usuário pressione a tecla **ENTER**. Só então o resto do programa é executado. Vejamos a saída de dados quando digitamos João como nome:

```
Digite seu nome:João
Você digitou João
Olá, João!
```

Execute o programa outras vezes digitando, por exemplo, 123 como nome. Observe que o programa não se importa com os valores digitados pelo usuário. Essa verificação deve ser feita por seu programa. Veremos como fazer isso em outro capítulo, como validação de dados.

### 3.7.1 Conversão da entrada de dados

A função **input** sempre retorna valores do tipo string, ou seja, não importa se digitamos apenas números, o resultado sempre é string. Para resolver esse pequeno problema, vamos utilizar a função **int** para converter o valor retornado em um número inteiro, e a função **float** para convertê-lo em número decimal ou de ponto flutuante. Vejamos outro exemplo usando essas funções, no qual devemos calcular o valor de um bônus por tempo de serviço:

```
anos = int(input("Anos de serviço: "))
valor_por_ano = float(input("Valor por ano: "))
bônus = anos * valor_por_ano
print(f"Bônus de R$ {bônus:5.2f}")
```

Vejamos o resultado se testarmos 10 anos e R$ 25 por ano na tela seguinte. Observe que escrevemos apenas 25, e não R$ 25. Isso porque 25 é um número, e R$ 25 é uma string. Por enquanto, não vamos misturar dois tipos de dados na mesma entrada de dados, para simplificar nossos programas.

```
Anos de serviço: 10
Valor por ano: 25
Bônus de R$ 250.00
```

Execute o programa novamente com outros valores. Experimente digitar uma letra em anos de serviço ou em valor por ano. Você receberá uma mensagem de erro, pois a conversão de letras em números não é automática. A próxima seção explica melhor o problema.

**Exercício 3.7** Faça um programa que peça dois números inteiros. Imprima a soma desses dois números na tela.

**Exercício 3.8** Escreva um programa que leia um valor em metros e o exiba convertido em milímetros.

**Exercício 3.9** Escreva um programa que leia a quantidade de dias, horas, minutos e segundos do usuário. Calcule o total em segundos.

**Exercício 3.10** Faça um programa que calcule o aumento de um salário. Ele deve solicitar o valor do salário e a porcentagem do aumento. Exiba o valor do aumento e do novo salário.

**Exercício 3.11** Faça um programa que solicite o preço de uma mercadoria e o percentual de desconto. Exiba o valor do desconto e o preço a pagar.

**Exercício 3.12** Escreva um programa que calcule o tempo de uma viagem de carro. Pergunte a distância a percorrer e a velocidade média esperada para a viagem.

**Exercício 3.13** Escreva um programa que converta uma temperatura digitada em °C em °F. A fórmula para essa conversão é:

$$F = \frac{9 \times C}{5} + 32$$

**Exercício 3.14** Escreva um programa que pergunte a quantidade de km percorridos por um carro alugado pelo usuário, assim como a quantidade de dias pelos quais o carro foi alugado. Calcule o preço a pagar, sabendo que o carro custa R$ 60 por dia e R$ 0,15 por km rodado.

**Exercício 3.15** Escreva um programa para calcular a redução do tempo de vida de um fumante. Pergunte a quantidade de cigarros fumados por dia e quantos anos ele já fumou. Considere que um fumante perde 10 minutos de vida a cada cigarro, e calcule quantos dias de vida um fumante perderá. Exiba o total em dias.

### 3.7.2 Erros comuns

A entrada de dados é um ponto frágil em nossos programas. Como não temos como prever o que o usuário vai digitar, temos de nos preparar para reconhecer os erros mais comuns. Vejamos um programa que lê três valores:

```
nome = input("Digite seu nome: ")
idade = int(input("Digite sua idade: "))
saldo = float(input("Digite o saldo da sua conta bancária: "))
print(nome)
print(idade)
print(saldo)
```

Agora vejamos o resultado desse programa quando todos os valores são digitados corretamente. Correto significa uma ausência de erros durante a função **input** ou durante a conversão do valor retornado pela função.

```
Digite seu nome: João
Digite sua idade: 42
Digite o saldo da sua conta bancária: 15756.34
João
42
15756.34
```

A tela seguinte mostra outro exemplo de entrada de dados bem-sucedida. Observe que, como utilizamos a função **float** para converter o saldo, mesmo inserindo 34 o valor foi convertido para 34.0.

```
Digite seu nome: Maria
Digite sua idade: 28
Digite o saldo da sua conta bancária: 34
Maria
28
34.0
```

Vejamos agora um exemplo de erro durante a entrada de dados. No caso, digitamos letras (abc) que não podem ser convertidas em um valor inteiro. Observe que o erro interrompe nosso programa, exibindo a linha em que ocorreu e o nome do erro. Nesse caso, o erro aconteceu na linha 2 e seu nome é `ValueError: invalid literal for int with base 10: 'abc'`:

```
Digite seu nome: Minduim
Digite sua idade: abc
Traceback (most recent call last):
```

```
  File "input/input2.py", line 2, in <module>
    idade = int(input("Digite sua idade: "))
ValueError: invalid literal for int() with base 10: 'abc'
```

Outro erro de conversão, mas, dessa vez, durante a conversão para número decimal, usando a função **float**. A entrada de dados é um pouco rústica, parando em caso de erro. Mais adiante, aprenderemos sobre exceções em Python e como tratar esse tipo de erro. Por enquanto, basta saber que não estamos validando a entrada e que nossos programas ainda são frágeis.

```
Digite seu nome: Juanito
Digite sua idade: 31
Digite o saldo da sua conta bancária: abc
Traceback (most recent call last):
  File "input/input2.py", line 3, in <module>
    saldo = float(input("Digite o saldo da sua conta bancária: "))
ValueError: could not convert string to float: abc
```

Exemplo de erro de conversão, utilizando a vírgula no lugar de ponto:

```
Digite seu nome: Mary
Digite sua idade: 25
Digite o saldo da sua conta bancária: 17,4
Traceback (most recent call last):
  File "input/input2.py", line 3, in <module>
    saldo = float(input("Digite o saldo da sua conta bancária: "))
ValueError: invalid literal for float(): 17,4
```

O erro mostrado é muito comum em países onde se usa a vírgula e não o ponto como separador entre a parte inteira e fracionária de um número. Em Python, você deve sempre digitar valores decimais usando o ponto, e não a vírgula como em português. Assim, `17,4` é um valor inválido, pois deveria ter sido digitado como `17.4`. Existem recursos em Python para resolver esse tipo de problema, mas ainda é cedo para abordarmos o assunto.

# CAPÍTULO 4

# Condições

*Executar ou não executar? Eis a questão...*

Nem sempre todas as linhas dos programas serão executadas. Muitas vezes, será mais interessante decidir que partes do programa devem ser executadas com base no resultado de uma condição. A base dessas decisões consistirá em expressões lógicas que permitam representar escolhas em programas.

## 4.1 if

As condições servem para selecionar quando uma parte do programa deve ser ativada e quando deve ser simplesmente ignorada. Em Python, a estrutura de decisão é o **if**. Seu formato é apresentado a seguir:

```
if <condição>:
    bloco verdadeiro
```

O **if** nada mais é que o nosso "se". Poderemos, então, entendê-lo em português da seguinte forma: se a condição for verdadeira, faça algo.

Vejamos um exemplo de programa que lê dois valores e imprime qual é o maior:

```
# Programa 4.1 - Lê dois valores e imprime qual é o maior
a = int(input("Primeiro valor: "))
b = int(input("Segundo valor: "))
if a > b: ❶
    print("O primeiro valor é o maior!") ❷
if b > a: ❸
    print("O segundo valor é o maior!") ❹
```

Em ❶, temos a condição `a > b`. Essa expressão será avaliada, e, se o seu resultado for verdadeiro, a linha ❷ será executada. Se for falso, a linha ❷ será ignorada. O mesmo acontece para a condição `b > a` da linha ❸. Se o seu resultado for verdadeiro, a linha ❹ será executada. Se for falso, ignorada.

A sequência de execução do programa é alterada de acordo com os valores digitados como o primeiro e o segundo valores. Digite o Programa 4.1 e execute-o duas vezes. Na primeira vez, digite um valor maior primeiro e um menor em segundo. Na segunda vez, inverta esses valores e verifique se a mensagem na tela também mudou.

Quando o primeiro valor é maior que o segundo, temos as seguintes linhas sendo executadas: ❶, ❷, ❸. Quando o primeiro valor é menor que o segundo, temos outra sequência: ❶, ❸, ❹. É importante entender que a linha com a condição em si é executada mesmo se o resultado da expressão for falso.

As linhas ❶ e ❸ foram terminadas com o símbolo dois pontos (:). Quando isso acontece, temos o anúncio de um bloco de linhas a seguir. Em Python, um bloco é representado deslocando-se o início da linha para a direita. O bloco continua até a primeira linha com deslocamento diferente. Para deslocar o texto à direita, utilize espaços, normalmente 4 espaços para cada nível de indentação.

Observe que começamos a escrever a linha ❷ alguns caracteres mais à direita da linha anterior ❶ que iniciou o bloco. Como a linha ❸ foi escrita mais à esquerda, dizemos que o bloco da linha ❷ foi terminado.

**TRÍVIA**

Python é uma das poucas linguagens de programação que utiliza o deslocamento do texto à direita (recuo) para marcar o início e o fim de um bloco. Outras linguagens contam com palavras especiais para isso, como `BEGIN` e `END`, em Pascal; ou as famosas chaves ({ e }), em C e Java.

**Exercício 4.1** Analise o Programa 4.1. Responda o que acontece se o primeiro e o segundo valores forem iguais? Explique.

Vejamos outro exemplo, em que solicitaremos a idade do carro do usuário e, em seguida, escreveremos novo se o carro tiver menos de três anos; ou velho, caso contrário.

```
# Programa 4.2 - Carro novo ou velho, dependendo da idade
idade = int(input("Digite a idade do seu carro: "))
if idade <= 3:
    print("Seu carro é novo") ❶
if idade > 3:
    print("Seu carro é velho") ❷
```

Execute o Programa 4.2 e verifique o que aparece na tela. Você pode executá-lo várias vezes com as seguintes idades: 1, 3 e 5. A primeira condição é `idade <= 3`. Essa condição decide se a linha com função **`print`** ❶ será ou não executada. Como é uma condição simples, podemos entender que só exibiremos a mensagem do carro novo para as idades 0, 1, 2 e 3. A segunda condição, `idade > 3`, é o inverso da primeira. Se você observar de perto, não há um só número que torne ambas verdadeiras ao mesmo tempo. A segunda decisão é responsável por decidir a impressão da mensagem do carro velho ❷.

Embora óbvio que um carro não poderia ter valores negativos como idade, o programa não trata desse problema. Vamos alterá-lo mais adiante para verificar valores inválidos.

**Exercício 4.2** Escreva um programa que pergunte a velocidade do carro de um usuário. Caso ultrapasse 80 km/h, exiba uma mensagem dizendo que o usuário foi multado. Nesse caso, exiba o valor da multa, cobrando R$ 5 por km acima de 80 km/h.

Um bloco de linhas em Python pode ter mais de uma linha, embora o último exemplo mostre apenas dois blocos com uma linha em cada. Se você precisar de duas ou mais no mesmo bloco, escreva essas linhas na mesma direção ou na mesma coluna da primeira linha do bloco. Isso basta para representá-lo.

Um problema comum é quando temos de pagar Imposto de Renda. Normalmente, pagamos o Imposto de Renda por faixa de salário. Imagine que, para salários menores que R$ 1.000,00, não teríamos imposto a pagar, ou seja, alíquota 0%. Para salários entre R$ 1.000,00 e R$ 3.000,00, pagaríamos 20%. Acima desses valores, a alíquota seria de 35%. Esse problema se pareceria muito com o anterior, salvo se o imposto não fosse cobrado diferentemente para cada faixa, ou seja, quem ganha R$ 4.000,00 tem os primeiros R$ 1.000,00 isentos de imposto; com o montante entre R$ 1.000,00 e R$ 3.000,00 pagando 20%, e o restante pagando os 35%. Vejamos a solução:

```
# Programa 4.3 - Cálculo do Imposto de Renda
salário = float(input("Digite o salário para cálculo do imposto: "))
base = salário ❶
imposto = 0
if base > 3000: ❷
    imposto = imposto + ((base - 3000) * 0.35) ❸
    base = 3000 ❹
if base > 1000: ❺
    imposto = imposto + ((base - 1000) * 0.20) ❻
print(f"Salário: R${salário:6.2f} Imposto a pagar: R${imposto:6.2f}")
```

O Programa 4.3 é bem interessante. Tente executá-lo algumas vezes e compare o valor impresso com o valor calculado por você. Rastreie o programa e tente entender o que ele faz antes de ler o parágrafo seguinte. Verifique o que acontece para salários de R$ 500,00, R$ 1.000,00 e R$ 1.500,00.

Em ❶ temos a variável `base` recebendo uma cópia de `salário`. Isso é necessário porque, quando atribuímos um novo valor para uma variável, o valor anterior é substituído (e perdido se não o guardarmos em outro lugar). Como vamos utilizar o valor do salário digitado para exibi-lo na tela, não podemos perdê-lo; por isso, a necessidade de uma variável auxiliar chamada aqui de `base`.

Em ❷ verificamos se a `base` é maior que R$ 3.000,00. Se verdadeiro, executamos as linhas ❸ e ❹. Em ❸, calculamos 35% do valor superior a R$ 3.000,00. O resultado é armazenado na variável `imposto`. Como essa variável contém o valor a pagar para essa quantia, atualizaremos o valor de base para R$ 3.000,00 ❹, pois o que ultrapassa esse valor já foi tarifado.

Em ❺ verificamos se o valor de base é maior que R$ 1.000,00, calculando 20% de imposto em ❻, caso verdadeiro.

Vejamos o rastreamento para um salário de R$ 500,00:

| salário | base | imposto |
|---|---|---|
| 500 | 500 | 0 |

Para um salário de R$ 1.500,00:

| salário | base | imposto |
|---|---|---|
| 1500 | 1500 | 0 |
| | | 100 |

Para um salário de R$ 3.000,00:

| salário | base | imposto |
|---|---|---|
| 3000 | 3000 | 0 |
| | | 400 |

Para um salário de R$ 5.000,00:

| salário | base | imposto |
|---|---|---|
| 5000 | ~~5000~~ | 0 |
| | 3000 | ~~700~~ |
| | | 1100 |

**Exercício 4.3** Escreva um programa que leia três números e que imprima o maior e o menor.

**Exercício 4.4** Escreva um programa que pergunte o salário do funcionário e calcule o valor do aumento. Para salários superiores a R$ 1.250,00, calcule um aumento de 10%. Para os inferiores ou iguais, de 15%.

## 4.2 else

Quando há problemas, como a mensagem do carro velho (Programa 4.3), na qual a segunda condição é simplesmente o inverso da primeira, podemos usar outra forma de **if** para simplificar os programas. Essa forma é a cláusula **else** para especificar o que fazer caso o resultado da avaliação da condição seja falso, sem precisarmos de um novo **if**. Vejamos como ficaria o programa reescrito para usar **else**:

```
# Programa 4.4 - Carro novo ou velho, dependendo da idade com else
idade = int(input("Digite a idade de seu carro: "))
if idade <= 3:
    print("Seu carro é novo")
else: ❶
    print("Seu carro é velho") ❷
```

Veja que em ❶ utilizamos ":" após **else**. Isso é necessário porque **else** inicia um bloco, da mesma forma que **if**. É importante notar que devemos escrever **else** na mesma coluna do **if**, ou seja, com o mesmo recuo. Assim, o interpretador reconhece que **else** se refere a um determinado **if**. Você obterá um erro caso não alinhe essas duas estruturas na mesma coluna.

A vantagem de usar **else** é deixar os programas mais claros, uma vez que podemos expressar o que fazer caso a condição especificada em **if** seja falsa. A linha ❷ só é executada se a condição `idade <= 3` for falsa.

**Exercício 4.5** Execute o Programa 4.4 e experimente alguns valores. Verifique se os resultados foram os mesmos do Programa 4.2.

**Exercício 4.6** Escreva um programa que pergunte a distância que um passageiro deseja percorrer em km. Calcule o preço da passagem, cobrando R$ 0,50 por km para viagens de até de 200 km, e R$ 0,45 para viagens mais longas.

## 4.3 Estruturas aninhadas

Nem sempre nossos programas serão tão simples. Muitas vezes, precisaremos aninhar vários **if** para obter o comportamento desejado do programa. Aninhar, nesse caso, é utilizar um **if** dentro de outro.

Vejamos o exemplo de calcular a conta de um telefone celular da empresa Tchau. Os planos da empresa Tchau são bem interessantes e oferecem preços diferenciados de acordo com a quantidade de minutos usados por mês. Abaixo de 200 minutos, a empresa cobra R$ 0,20 por minuto. Entre 200 e 400 minutos, o preço é de R$ 0,18. Acima de 400 minutos, o preço por minuto é de R$ 0,15. O Programa 4.5 resolve esse problema:

```
# Programa 4.5 - Conta de telefone com três faixas de preço
minutos = int(input("Quantos minutos você utilizou este mês:"))
if minutos < 200: ❶
    preço = 0.20 ❷
else:
    if minutos < 400: ❸
        preço = 0.18 ❹
    else: ❺
        preço = 0.15 ❻
print(f"Você vai pagar este mês: R${minutos * preço:6.2f}")
```

Em ❶, temos a primeira condição: `minutos < 200`. Se a quantidade de minutos for menor que 200, atribuímos 0,20 ao preço em ❷. Até aqui, nada de novo. Observe que **if** de ❸ está dentro de **else** da linha anterior: dizemos que está aninhado dentro de **else**. A condição de ❸, `minutos < 400`, decide se vamos executar a linha de ❹ ou a de ❻. Observe que **else** de ❺ está alinhado com **if** de ❸. No final, calculamos e imprimimos o preço na tela. Lembre-se de que o alinhamento do texto é muito importante em Python.

Vejamos, por exemplo, a situação em que cinco categorias são necessárias. Façamos um programa que leia a categoria de um produto e determine o preço pela Tabela 4.1.

*Tabela 4.1 – Categorias de produto e preço*

| Categoria | Preço |
|---|---|
| 1 | 10,00 |
| 2 | 18,00 |
| 3 | 23,00 |
| 4 | 26,00 |
| 5 | 31,00 |

Na coluna mais à esquerda do Programa 4.6, você encontrará os números de linha do programa, numeradas de 1 a 19. Esses números servem apenas para ajudar o entendimento da explicação a seguir: lembre-se de não os digitar.

```
# Programa 4.6 - Categoria x preço
1  categoria = int(input("Digite a categoria do produto:"))
2  if categoria == 1:
3          preço = 10
4  else:
5      if categoria == 2:
6          preço = 18
7      else:
8          if categoria == 3:
9              preço = 23
10         else:
11             if categoria == 4:
12                 preço = 26
13             else:
14                 if categoria == 5:
15                     preço = 31
```

```
16                else:
17                    print("Categoria inválida, digite um valor entre 1 e 5!")
18                    preço = 0
19 print(f"O preço do produto é: R${preço:6.2f}")
```

Observe que o alinhamento se tornou um grande problema, uma vez que tivemos de deslocar à direita a cada **else**.

No Programa 4.6, introduzimos o conceito de validação da entrada. Dessa vez, se o usuário digitar um valor inválido, receberá uma mensagem de erro na tela. Nada muito prático ou bonito.

Vejamos a execução das linhas dependendo da categoria digitada na Tabela 4.2.

*Tabela 4.2 – Linhas executadas*

| Categoria | Linhas executadas |
|---|---|
| 1 | 1,2,3,19 |
| 2 | 1,2,4,5,6,19 |
| 3 | 1,2,4,5,7,8,9,19 |
| 4 | 1,2,4,5,7,8,10,11,12,19 |
| 5 | 1,2,4,5,7,8,10,11,13,14,15,19 |
| outras | 1,2,4,5,7,8,10,11,13,14,16,17,18,19 |

Quando lermos um programa com estruturas aninhadas, devemos prestar muita atenção para visualizar corretamente os blocos. Observe como o alinhamento é importante.

**Exercício 4.7** Rastreie o Programa 4.6. Compare seu resultado ao apresentado na Tabela 4.2.

## 4.4 elif

Python apresenta uma solução muito interessante ao problema de múltiplos **if**s aninhados. A cláusula **elif** substitui um par **else if**, mas sem criar outro nível de estrutura, evitando problemas de deslocamentos desnecessários à direita.

Vamos revisitar o Programa 4.6, dessa vez usando **elif**. Veja o resultado no Programa 4.7:

```
# Programa 4.7 - Categoria x preço, usando elif
categoria = int(input("Digite a categoria do produto:"))
if categoria == 1:
    preço = 10
elif categoria == 2:
    preço = 18
elif categoria == 3:
    preço = 23
elif categoria == 4:
    preço = 26
elif categoria == 5:
    preço = 31
else:
    print("Categoria inválida, digite um valor entre 1 e 5!")
    preço = 0
print(f"O preço do produto é: R${preço:6.2f}")
```

**Exercício 4.8** Escreva um programa que leia dois números e que pergunte qual operação você deseja realizar. Você deve poder calcular soma (+), subtração (-), multiplicação (*) e divisão (/). Exiba o resultado da operação solicitada.

**Exercício 4.9** Escreva um programa para aprovar o empréstimo bancário para compra de uma casa. O programa deve perguntar o valor da casa a comprar, o salário e a quantidade de anos a pagar. O valor da prestação mensal não pode ser superior a 30% do salário. Calcule o valor da prestação como sendo o valor da casa a comprar dividido pelo número de meses a pagar.

**Exercício 4.10** Escreva um programa que calcule o preço a pagar pelo fornecimento de energia elétrica. Pergunte a quantidade de kWh consumida e o tipo de instalação: R para residências, I para indústrias e C para comércios. Calcule o preço a pagar de acordo com a tabela a seguir.

| Preço por tipo e faixa de consumo | | |
|---|---|---|
| Tipo | Faixa (kWh) | Preço |
| Residencial | Até 500 | R$ 0,40 |
| | Acima de 500 | R$ 0,65 |
| Comercial | Até 1000 | R$ 0,55 |
| | Acima de 1000 | R$ 0,60 |
| Industrial | Até 5000 | R$ 0,55 |
| | Acima de 5000 | R$ 0,60 |

# CAPÍTULO 5

# Repetições

Repetições representam a base de vários programas. São utilizadas para executar a mesma parte de um programa várias vezes, normalmente dependendo de uma condição. Por exemplo, para imprimir três números na tela, poderíamos escrever um programa como:

```
print(1)
print(2)
print(3)
```

Podemos imaginar que para imprimir três números, começando de 1 até o 3, devemos variar **print**(x), em que x varia de 1 a 3. Vejamos outra solução:

```
x = 1
print(x)
x = 2
print(x)
x = 3
print(x)
```

Outra solução seria incrementar o valor de x após cada **print**:

```
x = 1
print(x)
x = x + 1
print(x)
x = x + 1
print(x)
```

Porém, se o objetivo fosse escrever 100 números, a solução não seria tão agradável, pois teríamos de escrever pelo menos 200 linhas! A estrutura de repetição aparece para nos auxiliar a resolver esse tipo de problema.

Uma das estruturas de repetição do Python é o **while**, que repete um bloco enquanto a condição for verdadeira. Seu formato é apresentado a seguir, em

que `condição` é uma expressão lógica, e `bloco` representa as linhas de programa a repetir enquanto o resultado da condição for verdadeiro.

```
while <condição>:
    bloco
```

Para resolver o problema de escrever três números utilizando o **`while`**, escreveríamos um programa:

```
x = 1 ❶
while x <= 3: ❷
    print(x) ❸
    x = x + 1 ❹
```

A execução desse programa seria um pouco diferente do que vimos até agora. Primeiro, ❶ seria executada inicializando a variável `x` com o valor 1. A linha ❷ seria uma combinação de estrutura condicional com estrutura de repetição. Podemos entender a condição do **`while`** da mesma forma que a condição de **`if`**. A diferença é que, se a condição for verdadeira, repetiremos as linhas ❸ e ❹ (bloco) enquanto a avaliação da condição for verdadeira.

Em ❸ teremos a impressão na tela propriamente dita, em que `x` é 1. Em ❹ temos que o valor de `x` é acrescentado de 1. Como `x` vale 1, `x + 1` valerá 2. Esse novo valor é então atribuído a `x`. A parte nova é que a execução não termina após ❹, que é o fim do bloco, mas retorna para ❷. É esse retorno que faz a estrutura de repetição especial.

Agora, `x` vale 2 e `x <= 3` continua verdadeiro (**`True`**), logo, o bloco será executado outra vez. ❸ realizará a impressão do valor 2, e ❹ atualizará o valor de `x` para `x + 1`; nesse caso, 2 + 1 = 3. A execução volta novamente para a linha ❷.

A condição em ❷ é avaliada, e, como `x` vale 3, `x <= 3` continua verdadeira, fazendo com que as linhas ❸ e ❹ sejam executadas, exibindo 3 e atualizando o valor de `x` para 4 (3 + 1).

Nesse ponto, ❷, temos que `x` vale 4 e que a condição `x <= 3` resulta em Falso (**`False`**), terminando, assim, a repetição do bloco.

**Exercício 5.1** Modifique o programa para exibir os números de 1 a 100.

**Exercício 5.2** Modifique o programa para exibir os números de 50 a 100.

**Exercício 5.3** Faça um programa para escrever a contagem regressiva do lançamento de um foguete. O programa deve imprimir 10, 9, 8, ..., 1, 0 e Fogo! na tela.

## 5.1 Contadores

O poder das estruturas de repetições é muito interessante, principalmente quando utilizamos condições com mais de uma variável. Imagine um problema em que deveríamos imprimir os números inteiros entre 1 e um valor digitado pelo usuário. Vamos escrever um programa de forma que o último número a imprimir seja informado pelo usuário:

```
fim = int(input("Digite o último número a imprimir:")) ❶
x = 1
while x <= fim: ❷
    print(x) ❸
    x = x + 1 ❹
```

Nesse caso, o programa imprimirá de 1 até o valor digitado em ❶. Em ❷ utilizamos a variável `fim` para representar o limite de nossa repetição.

Agora vamos analisar o que realizamos com a variável `x` dentro da repetição. Em ❸, o valor de `x` é simplesmente impresso. Em ❹ atualizamos o valor de `x` com `x + 1`, ou seja, com o próximo valor inteiro. Quando realizamos esse tipo de operação dentro de uma repetição, estamos contando. Logo, diremos que `x` é um contador. Um contador é uma variável utilizada para contar o número de ocorrências de um determinado evento; nesse caso, o número de repetições do **`while`**, que satisfaz às necessidades de nosso problema.

Experimente esse programa com vários valores, primeiro digitando 5, depois 500 e, por fim, 0 (zero). Provavelmente, 5 e 500 produzirão os resultados esperados, ou seja, a impressão de 1 até 5, ou de 1 até 500. Porém, quando digitamos zero, nada acontece, e o programa termina logo a seguir, sem impressão.

Analisando nosso programa quando a variável `fim` vale 0, ou seja, quando digitamos 0 em ❶, temos que a condição em ❷ é `x <= fim`. Como `x` é 1 e `fim` é 0, temos que `1 <= 0` é falso desde a primeira execução, fazendo com que o bloco a repetir não seja executado, uma vez que sua condição de entrada é falsa. O mesmo aconteceria na inserção de valores negativos.

Imagine que o problema agora seja um pouco diferente: imprimir apenas os números pares entre 0 e um número digitado pelo usuário, de forma bem similar ao problema anterior. Poderíamos resolver o problema com um **`if`** para testar se `x` é par ou ímpar antes de imprimir. Vale lembrar que um número é par quando é 0 ou múltiplo de 2. Quando é múltiplo de 2, temos que o resto da divisão desse número por 2 é 0, ou seja, o resultado é uma divisão exata, sem resto.

Em Python, podemos escrever esse teste com `x % 2 == 0` (resto da divisão de x por 2 é igual a zero); alterando o programa anterior, temos:

```
fim = int(input("Digite o último número a imprimir:"))
x = 0 ❶
while x <= fim:
    if x % 2 == 0: ❷
        print(x) ❸
    x = x + 1
```

Veja que, para começar a imprimir do 0, e não de 1, modificamos ❶. Um detalhe importante é que ❸ é um bloco dentro de **if** ❷, sendo para isso deslocado à direita. Execute o programa e verifique seu resultado.

Agora, finalmente, estamos resolvendo o problema, mas poderíamos resolvê-lo de forma ainda mais simples se adicionássemos 2 a x a cada repetição. Isso garantiria que x sempre fosse par:

```
fim = int(input("Digite o último número a imprimir:"))
x = 0
while x <= fim:
    print(x)
    x = x + 2
```

Esses dois exemplos mostram que existe mais de uma solução para o problema, que podemos escrever programas diferentes e obter a mesma solução. Essas soluções podem ser às vezes mais complicadas, às vezes mais simples, mas ainda assim corretas.

**Exercício 5.4** Modifique o programa anterior para imprimir de 1 até o número digitado pelo usuário, mas, dessa vez, apenas os números ímpares.

**Exercício 5.5** Reescreva o programa anterior para escrever os 10 primeiros múltiplos de 3.

Vejamos outro tipo de problema. Imagine ter de imprimir a tabuada de adição de um número digitado pelo usuário. Essa tabuada deve ser impressa de 1 a 10, sendo n o número digitado pelo usuário. Teríamos, assim, n+1, n+2, ... n+10.

```
n = int(input("Tabuada de:"))
x = 1
while x <= 10:
    print(n + x)
    x = x + 1
```

Execute o programa anterior e experimente diversos valores.

**Exercício 5.6** Altere o programa anterior para exibir os resultados no mesmo formato de uma tabuada: 2x1 = 2, 2x2 = 4, ...

**Exercício 5.7** Modifique o programa anterior de forma que o usuário também digite o início e o fim da tabuada, em vez de começar com 1 e 10.

**Exercício 5.8** Escreva um programa que leia dois números. Imprima o resultado da multiplicação do primeiro pelo segundo. Utilize apenas os operadores de soma e subtração para calcular o resultado. Lembre-se de que podemos entender a multiplicação de dois números como somas sucessivas de um deles. Assim, 4 × 5 = 5 + 5 + 5 + 5 = 4 + 4 + 4 + 4 + 4.

**Exercício 5.9** Escreva um programa que leia dois números. Imprima a divisão inteira do primeiro pelo segundo, assim como o resto da divisão. Utilize apenas os operadores de soma e subtração para calcular o resultado. Lembre-se de que podemos entender o quociente da divisão de dois números como a quantidade de vezes que podemos retirar o divisor do dividendo. Logo, 20 ÷ 4 = 5, uma vez que podemos subtrair 4 cinco vezes de 20.

Contadores também podem ser úteis quando usados com condições dentro dos programas. Vejamos um programa para corrigir um teste de múltipla escolha com três questões. A resposta da primeira é "b"; da segunda, "a"; e da terceira, "d". O programa seguinte conta um ponto a cada resposta correta:

```
pontos = 0
questão = 1
while questão <= 3:
    resposta = input(f"Resposta da questão {questão}: ")
    if questão == 1 and resposta == "b":
        pontos = pontos + 1
    if questão == 2 and resposta == "a":
        pontos = pontos + 1
    if questão == 3 and resposta == "d":
        pontos = pontos + 1
    questão = questão + 1
print(f"O aluno fez {pontos} ponto(s)")
```

Execute o programa e digite todas as respostas corretas, depois tente com respostas diferentes. Veja que estamos verificando apenas respostas simples de uma só letra e que consideramos apenas letras minúsculas. Em Python, uma letra minúscula é diferente de uma maiúscula. Se você digitar "A" na segunda questão em vez de "a", o programa não considerará essa resposta correta. Uma solução para esse

tipo de problema é utilizar o operador lógico **or** e verificar a resposta maiúscula e minúscula. Por exemplo, `questão == 1 and (resposta == "b" or resposta == "B")`.

**Exercício 5.10** Modifique o programa anterior para que aceite respostas com letras maiúsculas e minúsculas em todas as questões.

Embora essa verificação resolva o problema, veremos que, se digitarmos um espaço em branco antes ou depois da resposta, ela também será considerada errada. Sempre que trabalharmos com strings, esse tipo de problema deve ser controlado. Veremos mais sobre o assunto no Capítulo 7.

## 5.2 Acumuladores

Nem só de contadores precisamos. Em programas para calcular o total de uma soma, por exemplo, precisaremos de acumuladores. A diferença entre um contador e um acumulador é que nos contadores o valor adicionado é constante e, nos acumuladores, variável. Vejamos um programa que calcule a soma de 10 números. Nesse caso, `soma` ❶ é um acumulador e `n` ❷ é um contador.

```
n = 1
soma = 0
while n <= 10:
    x = int(input(f"Digite o {n} número:"))
    soma = soma + x ❶
    n = n + 1 ❷
print(f"Soma: {soma}")
```

Podemos definir a média aritmética como a soma de vários números divididos pela quantidade de números somados. Assim, se somarmos três números, 4, 5 e 6, teríamos a média aritmética como (4+5+6) / 3, em que 3 é a quantidade de números. Se chamarmos o primeiro número de n1, o segundo de n2, e o terceiro de n3, teremos (n1 + n2 + n3) / 3.

Vejamos um programa que calcula a média de cinco números digitados pelo usuário. Se chamarmos o primeiro valor digitado de n1, o segundo de n2, e assim sucessivamente, teremos que:

$$média = (n1+n2+n3+n4+n5)/5 = \frac{n1 + n2 + n3 + n4 + n5}{5}$$

Em vez de utilizarmos cinco variáveis, vamos acumular os valores à medida que são lidos.

```
x = 1
soma = 0 ❶
while x <= 5:
    n = int(input(f"{x} Digite o número:"))
    soma = soma + n ❷
    x = x + 1
print("Média: {soma / 5:5.2f}") ❸
```

Nesse caso, temos x sendo um contador e n o valor digitado pelo usuário. A variável soma é criada em ❶ e inicializada com 0. Diferentemente de x, que recebe 1 a cada passagem, a variável soma, em ❷, é adicionada do valor digitado pelo usuário. Podemos dizer que o incremento de soma não é um valor constante, pois varia com o valor digitado pelo usuário. Podemos também dizer que soma acumula os valores de n a cada repetição. Logo, diremos que a variável soma é um acumulador.

Acumuladores são muito interessantes quando não sabemos ou não conseguimos obter o total da soma pela simples multiplicação de dois números. No caso do cálculo da média, o valor de n pode ser diferente cada vez que o usuário digitar um valor.

**Exercício 5.11** Escreva um programa que pergunte o depósito inicial e a taxa de juros de uma poupança. Exiba os valores mês a mês para os 24 primeiros meses. Escreva o total ganho com juros no período.

**Exercício 5.12** Altere o programa anterior de forma a perguntar também o valor depositado mensalmente. Esse valor será depositado no início de cada mês, e você deve considerá-lo para o cálculo de juros do mês seguinte.

**Exercício 5.13** Escreva um programa que pergunte o valor inicial de uma dívida e o juro mensal. Pergunte também o valor mensal que será pago. Imprima o número de meses para que a dívida seja paga, o total pago e o total de juros pago.

### 5.2.1 Operadores de atribuição especiais

Muitas vezes, teremos de escrever expressões como x = x + 1 ou y = y - 1. Para simplificar a escrita, a linguagem Python oferece operadores de atribuição especiais como += e -=. Esses operadores têm o mesmo significado de expressões com os sinais que o precedem. Por exemplo:

*Tabela 5.1 – Operadores de atribuição especiais*

| Operador | Exemplo | Equivalência |
|---|---|---|
| += | x += 1 | x = x + 1 |
| -= | y -= 1 | y = y - 1 |
| *= | c *= 2 | c = c * 2 |
| /= | d /= 2 | d = d / 2 |
| **= | e **= 2 | e = e ** 2 |
| //= | f //= 4 | f = f // 4 |

## 5.3 Interrompendo a repetição

Embora muito útil, a estrutura **while** só verifica sua condição de parada no início de cada repetição. Dependendo do problema, a habilidade de terminar **while** dentro do bloco a repetir pode ser interessante.

A instrução **break** é utilizada para interromper a execução de **while** independentemente do valor atual de sua condição. Vejamos o exemplo da leitura de valores até que digitemos 0 (zero):

```
s = 0
while True: ❶
    v = int(input("Digite um número a somar ou 0 para sair:"))
    if v == 0:
        break ❷
    s += v ❸
print(s) ❹
```

Nesse exemplo, substituímos a condição do **while** por **True** em ❶. Dessa forma, o **while** executará para sempre, pois o valor de sua condição de parada (**True**) é constante. Em ❷ temos a instrução **break** sendo ativada dentro de um **if**, especificamente quando v é zero. Porém, enquanto v for diferente de zero, a repetição continuará a somar v a s em ❸. Quando v for igual a zero (0), teremos ❷ sendo executada, terminando a repetição e transferindo a execução para ❹, que, então, exibe o valor de s na tela.

**Exercício 5.14** Escreva um programa que leia números inteiros do teclado. O programa deve ler os números até que o usuário digite 0 (zero). No final da execução, exiba a quantidade de números digitados, assim como a soma e a média aritmética.

**Exercício 5.15** Escreva um programa para controlar uma pequena máquina registradora. Você deve solicitar ao usuário que digite o código do produto e a quantidade comprada. Utilize a tabela de códigos a seguir para obter o preço de cada produto:

| Código | Preço |
|---|---|
| 1 | 0,50 |
| 2 | 1,00 |
| 3 | 4,00 |

| Código | Preço |
|---|---|
| 5 | 7,00 |
| 9 | 8,00 |

Seu programa deve exibir o total das compras depois que o usuário digitar 0. Qualquer outro código deve gerar a mensagem de erro "Código inválido".

Vejamos como exemplo um programa que leia um valor e que imprima a quantidade de cédulas necessárias para pagar esse mesmo valor. Para simplificar, vamos trabalhar apenas com valores inteiros e com cédulas de R$ 50, R$ 20, R$ 10, R$ 5 e R$ 1.

```
# Programa 5.1 - Contagem de cédulas
valor = int(input("Digite o valor a pagar:"))
cédulas = 0
atual = 50
apagar = valor
while True:
    if atual <= apagar:
        apagar -= atual
        cédulas += 1
    else:
        print(f"{cédulas} cédula(s) de R${atual}")
        if apagar == 0:
            break
        if atual == 50:
            atual = 20
        elif atual == 20:
            atual = 10
        elif atual == 10:
            atual = 5
        elif atual == 5:
            atual = 1
        cédulas = 0
```

**Exercício 5.16** Execute o Programa 5.1 para os seguintes valores: 501, 745, 384, 2, 7 e 1.

**Exercício 5.17** O que acontece se digitarmos 0 (zero) no valor a pagar?

**Exercício 5.18** Modifique o programa para também trabalhar com notas de R$ 100.

**Exercício 5.19** Modifique o programa para aceitar valores decimais, ou seja, também contar moedas de 0,01, 0,02, 0,05, 0,10 e 0,50.

**Exercício 5.20** O que acontece se digitarmos 0,001 no programa anterior? Caso ele não funcione, altere-o de forma a corrigir o problema.

## 5.4 Repetições aninhadas

Podemos combinar vários **while** de forma a obter resultados mais interessantes, como a repetição com incremento de duas variáveis. Imagine imprimir as tabuadas de multiplicação de 1 a 10. Vejamos como fazer isso:

```
tabuada = 1
while tabuada <= 10: ❶
    número = 1 ❷
    while número <= 10: ❸
        print(f"{tabuada} x {número} = {tabuada * número}")
        número += 1 ❹
    tabuada += 1 ❺
```

Em ❶ temos nosso primeiro **while**, criado para repetir seu bloco enquanto o valor de `tabuada` for menor ou igual a 10. Em ❷ temos a inicialização da variável `número` dentro do primeiro **while**. Isso é importante porque precisamos voltar a multiplicar por 1 a cada novo valor da variável `tabuada`. Finalmente, em ❸ temos o segundo **while** com a condição de parada `número <= 10`. Esse **while** executará suas repetições dentro do primeiro, ou seja, o ponto de execução passa de ❹ para ❸ enquanto a condição for verdadeira. Veja que em ❹ utilizamos o operador `+=` para representar `número = número + 1`. Quando `número` valer 11, a condição em ❸ resultará falsa, e a execução do programa continuará a partir da linha ❺. Em ❺ incrementamos o valor de `tabuada` e voltamos a ❶, onde será verificada a condição do primeiro **while**. Como resulta verdadeiro, voltaremos a executar ❷, reinicializando a variável `número` com o valor 1. ❷ é muito importante para que possamos novamente executar o segundo **while**, responsável por imprimir a tabuada na tela.

Vejamos o mesmo problema, mas sem utilizar repetições aninhadas:

```
tabuada = 1
número = 1
while tabuada <= 10:
    print(f"{tabuada} x {número} = {tabuada * número}")
    número += 1
    if número == 11:
        número = 1
        tabuada += 1
```

**Exercício 5.21** Reescreva o Programa 5.1 de forma a continuar executando até que o valor digitado seja 0. Utilize repetições aninhadas.

**Exercício 5.22** Escreva um programa que exiba uma lista de opções (menu): adição, subtração, divisão, multiplicação e sair. Imprima a tabuada da operação escolhida. Repita até que a opção saída seja escolhida.

**Exercício 5.23** Escreva um programa que leia um número e verifique se é ou não um número primo. Para fazer essa verificação, calcule o resto da divisão do número por 2 e depois por todos os números ímpares até o número lido. Se o resto de uma dessas divisões for igual a zero, o número não é primo. Observe que 0 e 1 não são primos e que 2 é o único número primo que é par.

**Exercício 5.24** Modifique o programa anterior de forma a ler um número `n`. Imprima os `n` primeiros números primos.

**Exercício 5.25** Escreva um programa que calcule a raiz quadrada de um número. Utilize o método de Newton para obter um resultado aproximado. Sendo n o número a obter a raiz quadrada, considere a base b=2. Calcule p usando a fórmula p=(b+(n/b))/2. Agora, calcule o quadrado de p. A cada passo, faça b=p e recalcule p usando a fórmula apresentada. Pare quando a diferença absoluta entre n e o quadrado de p for menor que 0,0001.

**Exercício 5.26** Escreva um programa que calcule o resto da divisão inteira entre dois números. Utilize apenas as operações de soma e subtração para calcular o resultado.

**Exercício 5.27** Escreva um programa que verifique se um número é palíndromo. Um número é palíndromo se continua o mesmo caso seus dígitos sejam invertidos. Exemplos: 454, 10501

## 5.5 F-Strings

Já utilizamos F-Strings para compor strings, mas vamos revisitar o assunto. F-Strings foram introduzidas na versão 3.6 do Python. Com essa sintaxe, é possível substituir o valor de uma variável ou expressão dentro de uma string. Por exemplo:

```
>>> a = "mundo"
>>> print(f"Alô {a}")
Alô mundo
```

Em que `f"Alô {a}"` é equivalente a `"Alô %s" % a` ou `"Alô {}".format(a)`.

Você pode também formatar f-strings especificando o número de caracteres após o nome da variável e dos dois pontos. Exemplo:

```
>>> preço = 5.20
>>> f"Preço: {preço:5.2f}"
Preço:  5.20
>>> f"Preço: {preço:10.2f}"
Preço:       5.20
>>> f"Preço: R${preço:10.2f}"
Preço: R$      5.20
>>> f"Preço: R${preço:.2f}"
Preço: R$5.20
```

Você também pode usar >, < e ^ para alinhar os valores à esquerda, à direita ou ao centro:

```
>>> f"Preço: R${preço:>10.2f}"
Preço: R$      5.20
>>> f"Preço: R${preço:<10.2f}!"
Preço: R$5.20      !
>>> f"Preço: R${preço:^10.2f}!"
Preço: R$   5.20   !
```

E também especificar qual caractere deve ser utilizado para preencher os espaços em branco:

```
>>> f"Preço: R${preço:.^10.2f}!"
Preço: R$...5.20...!
>>> f"Preço: R${preço:x^10.2f}!"
Preço: R$xxx5.20xxx!
>>> f"Preço: R${preço:_^10.2f}!"
Preço: R$___5.20___!
```

Essa nova forma de escrever é tão poderosa que você pode até chamar funções dentro da f-string:

```
>>> x = 5.1
>>> f"Inteiro: {int(x)}"
'Inteiro: 5'
```

Assim como realizar operações matemáticas:

```
>>> f"Preço: R${preço * 10:5.2f}!"
Preço: R$52.00!
```

F-strings também funcionam com strings de múltiplas linhas, usando-se as aspas triplas prefixadas com a letra f:

```
>>> f"""
... O preço do novo produto é: R${preço:5.2f}.
... E pode ser encontrado nas melhores lojas do ramo.
... """
'"\nO preço do novo produto é: R$ 5.20.\nE pode ser encontrado nas melhores lojas do
   ramo.\n'
```

Observe que as linhas foram representadas com \n. Esta combinação de caracteres é utilizada para representar uma quebra de linha e você pode utilizá-la em suas strings:

```
>>> a = "primeira linha\nsegunda linha\nterceira linha"
>>> a
'primeira linha\nsegunda linha\nterceira linha'
>>> print(a)
primeira linha
segunda linha
terceira linha
```

NOTA: Qual formato você deve utilizar depende da sua aplicação. Se você trabalha com Python 3.6 ou superior, utilize f-strings. Se seu programa precisa rodar em outras versões de Python, anteriores à versão 3.6, utilize **format**. A utilização da formatação com % vem caindo em desuso, embora continue válida.

# CAPÍTULO 6

# Listas, dicionários, tuplas e conjuntos

Listas são um tipo de variável que permite o armazenamento de vários valores, acessados por um índice. Uma lista pode conter zero ou mais elementos de um mesmo tipo ou de tipos diversos, podendo inclusive conter outras listas. O tamanho de uma lista é igual à quantidade de elementos que ela contém.

Podemos imaginar uma lista como um edifício de apartamentos, em que o térreo é o andar zero, o primeiro andar é o andar 1 e assim por diante. O índice é utilizado para especificarmos o "apartamento" onde guardaremos nossos dados.

Em um prédio de seis andares, teremos números de andar variando entre 0 e 5. Se chamarmos nosso prédio de `P`, teremos `P[0]` como o endereço do térreo, `P[1]` como endereço do primeiro andar, continuando assim até `P[5]`. Em Python, `P` seria o nome da lista; e o número entre colchetes, o índice.

Listas são mais flexíveis que prédios e podem crescer ou diminuir com o tempo. Vejamos como criar uma lista em Python:

```
L = []
```

Essa linha cria uma lista chamada `L` com zero elemento, ou seja, uma lista vazia. Os colchetes (`[]`) após o símbolo de igualdade servem para indicar que `L` é uma lista. Vejamos agora como criar uma lista `Z`, com 3 elementos:

```
Z = [15, 8, 9]
```

A lista `Z` foi criada com três elementos: 15, 8 e 9. Dizemos que o tamanho da lista `Z` é 3. Como o primeiro elemento tem índice 0, temos que o último elemento é `Z[2]`. Veja o resultado de testes com `Z`:

```
>>> Z = [15, 8, 9]
>>> Z[0]
15
```

```
>>> Z[1]
8
>>> Z[2]
9
```

Utilizando o nome da lista e um índice, podemos mudar o conteúdo de um elemento. Observe os testes no interpretador:

```
>>> Z = [15, 8, 9]
>>> Z[0]
15
>>> Z[0] = 7
>>> Z[0]
7
>>> Z
[7, 8, 9]
```

Quando criamos a lista `Z`, o primeiro elemento era o número 15. Por isso, `Z[0]` era 15. Quando executamos `Z[0] = 7`, alteramos o conteúdo do primeiro elemento para 7. Isso pode ser verificado quando pedimos para exibir `Z`, agora com 7, 8 e 9 como elementos.

Vejamos um exemplo em que um aluno tem cinco notas e desejamos calcular a média aritmética dele:

```
# Programa 6.1 - Cálculo da média
notas = [6, 7, 5, 8, 9] ❶
soma = 0
x = 0
while x < 5: ❷
    soma += notas[x] ❸
    x += 1 ❹
print(f"Média: {soma / x:5.2f}")
```

Criamos a lista de notas em ❶. Em ❷, criamos a estrutura de repetição para variar o valor de `x` e continuar enquanto este for menor que 5. Lembre-se de que uma lista de cinco elementos contém índices de 0 a 4. Por isso inicializamos `x = 0` na linha anterior. Em ❸, adicionamos o valor de `notas[0]` à soma e depois `notas[1]`, `notas[2]`, `notas[3]` e `notas[4]`, um elemento a cada repetição. Para isso, utilizamos o valor de `x` como índice e o incrementamos de 1 em ❹. Uma grande vantagem desse programa foi que não precisamos declarar cinco variáveis para guardar as cinco notas. Todas as notas foram armazenadas na lista, utilizando um índice para identificar ou acessar cada valor.

Vejamos uma modificação desse exemplo, mas, dessa vez, vamos ler as notas uma a uma:

```
# Programa 6.2 - Cálculo da média com notas digitadas
notas = [0, 0, 0, 0, 0] ❶
soma = 0
x = 0
while x < 5:
    notas[x] = float(input(f"Nota {x}:")) ❷
    soma += notas[x]
    x += 1
x = 0 ❹
while x < 5: ❺
    print(f"Nota {x}: {notas[x]:6.2f}")
    x += 1
print(f"Média: {soma / x:5.2f}")
```

Em ❶ criamos a lista de notas com cinco elementos, todos zero. Em ❷, utilizamos a repetição para ler as notas do aluno e armazená-las na lista de notas. Veja que adicionamos `nota[x]` à `soma` já na linha seguinte. Terminada a primeira repetição, teremos a lista de notas preenchidas. Para imprimir a lista de notas, reinicializamos o valor da variável `x` para 0 ❸ e criamos outra estrutura de repetição ❹.

**Exercício 6.1** Modifique o Programa 6.2 para ler 7 notas em vez de 5.

## 6.1 Trabalhando com índices

Vejamos outro exemplo: um programa que lê cinco números, armazena-os em uma lista e depois solicita ao usuário que escolha um número a mostrar. O objetivo é, por exemplo, ler 15, 12, 5, 7 e 9 e armazená-los na lista. Depois, se o usuário digitar 2, ele imprimirá o segundo número digitado, 3, o terceiro, e assim sucessivamente. Observe que o índice do primeiro número é 0, e não 1: essa pequena conversão será feita no programa:

```
# Programa 6.3 - Apresentação de números
números = [0, 0, 0, 0, 0]
x = 0
while x < 5:
    números[x] = int(input(f"Número {x + 1}:")) ❶
    x += 1
```

```
while True:
    escolhido = int(input("Que posição você quer imprimir (0 para sair): "))
    if escolhido == 0:
        break
    print(f"Você escolheu o número: {números[escolhido - 1]}") ❷
```

Execute o Programa 6.3 e experimente alguns valores. Observe que em ❶ adicionamos 1 a `x` para que possamos imprimir `Número 1...5`, e não a partir de 0. Isso é importante porque começar a contar de 0 não é natural para a maioria das pessoas. Veja que, mesmo imprimindo `x + 1` para o usuário, a atribuição é feita para `números[x]` porque nossas listas começam em 0. Em ❷, fizemos a operação inversa. Quando o usuário escolhe o número a imprimir, ele faz uma escolha entre 1 e 5. Como 1 é o elemento 0, 2, o elemento 1, e assim por diante, diminuímos o valor da escolha de um para obtermos o índice de notas.

## 6.2 Cópia e fatiamento de listas

Embora listas em Python sejam um recurso muito poderoso, todo poder traz responsabilidades. Um dos efeitos colaterais de listas aparece quando tentamos fazer cópias. Vejamos um teste no interpretador:

```
>>> L = [1, 2, 3, 4, 5]
>>> V = L
>>> L
[1, 2, 3, 4, 5]
>>> V
[1, 2, 3, 4, 5]
>>> V[0] = 6
>>> V
[6, 2, 3, 4, 5]
>>> L
[6, 2, 3, 4, 5]
```

Veja que, ao modificarmos `V`, modificamos também o conteúdo de `L`. Isso porque uma lista em Python é um objeto e, quando atribuímos um objeto a outro, estamos apenas copiando a mesma referência da lista, e não seus dados em si. Nesse caso, `V` funciona como um apelido de `L`, ou seja, `V` e `L` são a mesma lista. Vejamos o que acontece no gráfico da Figura 6.1.

Quando modificamos `V[0]`, estamos modificando o mesmo valor de `L[0]`, pois ambos são referências, ou apelidos para a mesma lista na memória.

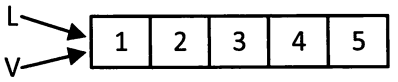


*Figura 6.1 – Duas variáveis referenciando a mesma lista.*

Dependendo da aplicação, esse efeito pode ser desejado ou não. Para criar uma cópia independente de uma lista, utilizaremos outra sintaxe. Vejamos o resultado das operações:

```
>>> L = [1, 2, 3, 4, 5]
>>> V = L[:]
>>> V[0] = 6
>>> L
[1, 2, 3, 4, 5]
>>> V
[6, 2, 3, 4, 5]
```

Ao escrevermos `L[:]`, estamos nos referindo a uma nova cópia de `L`. Assim `L` e `V` se referem a áreas diferentes na memória, permitindo alterá-las de forma independente, como na Figura 6.2.

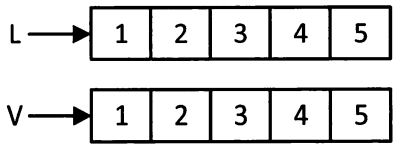


*Figura 6.2 – Duas variáveis referenciando duas listas.*

Podemos também fatiar uma lista, da mesma forma que fizemos com strings no Capítulo 3. Vejamos alguns exemplos no interpretador:

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0:5]
[1, 2, 3, 4, 5]
>>> L[:5]
[1, 2, 3, 4, 5]
>>> L[:-1]
[1, 2, 3, 4]
>>> L[1:3]
[2, 3]
>>> L[1:4]
[2, 3, 4]
```

```
>>> L[3:]
[4, 5]
>>> L[:3]
[1, 2, 3]
>>> L[-1]
5
>>> L[-2]
4
```

Veja que índices negativos também funcionam. Um índice negativo começa a contar do último elemento, mas observe que começamos de `-1`. Assim `L[0]` representa o primeiro elemento; `L[-1]`, o último; `L[-2]`, o penúltimo, e assim por diante.

## 6.3 Tamanho de listas

Podemos usar a função **`len`** com listas. O valor retornado é igual ao número de elementos da lista. Veja alguns testes:

```
>>> L = [12, 9, 5]
>>> len(L)
3
>>> V = []
>>> len(V)
0
```

A função **`len`** pode ser utilizada em repetições para controlar o limite dos índices:

```
# Programa 6.4 - Repetição com tamanho fixo da lista
L = [1, 2, 3]
x = 0
while x < 3:
    print(L[x])
    x += 1
```

Isso pode ser reescrito como:

```
# Programa 6.5 - Repetição com tamanho da lista usando len
L = [1, 2, 3]
x = 0
while x < len(L):
    print(L[x])
    x += 1
```

A vantagem é que se trocarmos L para:

```
L = [7, 8, 9, 10, 11, 12]
```

o resto do programa continuaria funcionando, pois utilizamos a função **len** para calcular o tamanho da lista. Observe que o valor retornado pela função **len** é um número que não pode ser utilizado como índice, mas que é perfeito para testarmos os limites de uma lista, como fizemos no Programa 6.5. Isso acontece porque **len** retorna a quantidade de elementos na lista e nossos índices começam a ser numerados de 0 (zero). Assim, os índices válidos de uma lista (L) variam de 0 até o valor de **len**(L) - 1.

## 6.4 Adição de elementos

Uma das principais vantagens de trabalharmos com listas é poder adicionar novos elementos durante a execução do programa. Para adicionar um elemento ao fim da lista, utilizaremos o método append. Em Python, chamamos um método escrevendo o nome dele após o nome do objeto. Como listas são objetos, sendo L a lista, teremos L.append(valor). Métodos são recursos de orientação a objetos, suportados e muito usados em Python. Você pode imaginar um método como uma função do objeto. Quando invocado, ele já sabe a que objeto estamos nos referindo, pois o informamos à esquerda do ponto. Falaremos mais sobre métodos no Capítulo 10; por enquanto, observe como os utilizamos e saiba que são diferentes de funções. Vejamos um teste no interpretador:

```
>>> L = []
>>> L.append("a")
>>> L
['a']
>>> L.append("b")
>>> L
['a', 'b']
>>> L.append("c")
>>> L
['a', 'b', 'c']
>>> len(L)
3
```

Vejamos um programa que lê números até que 0 seja digitado. Esse programa depois os imprimirá na mesma ordem em que foram digitados:

```
# Programa 6.6 - Adição de elementos à lista
L = []
while True:
    n = int(input("Digite um número (0 sai):"))
    if n == 0:
        break
    L.append(n)
x = 0
while x < len(L):
    print(L[x])
    x += 1
```

Esse simples programa é capaz de ler e imprimir um número inicialmente indeterminado de valores. Isso é possível porque adicionamos elementos à lista `L`, conforme necessário.

Outra forma de adicionarmos elementos a uma lista é com adição de listas, usando +:

```
>>> L = []
>>> L = L + [1]
>>> L
[1]
>>> L += [2]
>>> L
[1, 2]
>>> L += [3, 4, 5]
>>> L
[1, 2, 3, 4, 5]
```

Quando adicionamos apenas um elemento, tanto `L.append(1)` quanto `L + [1]` produzem o mesmo resultado. Perceba que em `L + [1]` escrevemos o elemento a adicionar dentro de uma lista (`[1]`), e que, em `append`, apenas 1. Isso porque, quando adicionamos uma lista a outra, o interpretador executa um método chamado `extend`, que adiciona os elementos de uma lista a outra:

```
>>> L = ["a"]
>>> L.append("b")
>>> L
['a', 'b']
>>> L.extend(["c"])
>>> L
['a', 'b', 'c']
```

```
>>> L.append(["d", "e"])
>>> L
['a', 'b', 'c', ['d', 'e']]
>>> L.extend(["f", "g", "h"])
>>> L
['a', 'b', 'c', ['d', 'e'], 'f', 'g', 'h']
```

O método `extend` sequer aceita parâmetros que não sejam listas. Se você utilizar o método `append` com uma lista como parâmetro, em vez de adicionar os elementos no fim da lista, `append` adicionará a lista inteira, mas como apenas um novo elemento. Teremos então listas dentro de listas:

```
>>> L = ["a"]
>>> L.append(["b"])
>>> L.append(["c", "d"])
>>> len(L)
3
>>> L[1]
['b']
>>> L[2]
['c', 'd']
>>> len(L[2])
2
>>> L[2][1]
'd'
```

Esse conceito é interessante, pois permite a utilização de estruturas de dados mais complexas, como matrizes, árvores e registros. Por enquanto, vamos utilizar esse recurso para armazenar múltiplos valores por elemento.

**Exercício 6.2** Faça um programa que leia duas listas e que gere uma terceira com os elementos das duas primeiras.

**Exercício 6.3** Faça um programa que percorra duas listas e gere uma terceira sem elementos repetidos.

## 6.5 Remoção de elementos da lista

Como o tamanho da lista pode variar, permitindo a adição de novos elementos, podemos também retirar alguns elementos da lista, ou mesmo todos eles. Para isso, utilizaremos a instrução **`del`**:

```
>>> L = ["a", "b", "c"]
>>> del L[1]
>>> L
['a', 'c']
>>> del L[0]
>>> L
['c']
```

É importante notar que o elemento excluído não ocupa mais lugar na lista, fazendo com que os índices sejam reorganizados, ou melhor, que passem a ser calculados sem esse elemento.

Podemos também apagar fatias inteiras de uma só vez:

```
>>> L = list(range(101))
>>> del L[1:99]
>>> L
[0, 99, 100]
```

A função **range** gera uma sequência de elementos, mas um a cada chamada, sendo o que chamamos de generator. A função **list** é utilizada para transformar o resultado dessa função em uma lista. Veja mais detalhes sobre **range** na Seção 6.21.

## 6.6 Usando listas como filas

Uma lista pode ser utilizada como fila se obedecermos a certas regras de inclusão e eliminação de elementos. Em uma fila, a inclusão é sempre realizada no fim, e as remoções são feitas no início. Dizemos que o primeiro a chegar é o primeiro a sair (*FIFO – First In First Out*).

É mais simples de entender se imaginarmos uma fila de banco. Quando a agência abre pela manhã, a fila está vazia. Quando os clientes começam a chegar, eles vão diretamente para o fim da fila. Os caixas então começam a atender esses clientes por ordem de chegada, ou seja, o cliente que chegou primeiro será atendido primeiro. Uma vez que o cliente é atendido, ele sai da fila. Então, um novo cliente passa a ser o primeiro da fila e o próximo a ser atendido.

Para escrevermos algo similar em Python, imaginaremos uma lista de clientes, representando a fila, em que o valor de cada elemento é igual à ordem de chegada do cliente. Vamos imaginar uma lista inicial com 10 clientes. Se outro cliente chegar, realizaremos um `append` para que ele seja inserido no fim da fila (`fila.append(último)`). Para retirarmos um cliente da fila e atendê-lo, poderíamos fazer **del** `fila[0]`, porém, isso apagaria o cliente da fila. Se quisermos retirá-lo da

fila e, ao mesmo tempo, obter o elemento retirado, podemos utilizar o método pop `fila.pop(0)`. O método pop retorna o valor do elemento e o exclui da fila. Passamos 0 como parâmetro para indicar que queremos excluir o primeiro elemento. Veja o programa completo:

```
# Programa 6.7 - Simulação de uma fila de banco
último = 10
fila = list(range(1, último + 1))
while True:
    print(f"\nExistem {len(fila)} clientes na fila")
    print(f"Fila atual: {fila}")
    print("Digite F para adicionar um cliente ao fim da fila,")
    print("ou A para realizar o atendimento. S para sair.")
    operação = input("Operação (F, A ou S):")
    if operação == "A":
        if len(fila) > 0:
            atendido = fila.pop(0)
            print(f"Cliente {atendido} atendido")
        else:
            print("Fila vazia! Ninguém para atender.")
    elif operação == "F":
        último += 1  # Incrementa o ticket do novo cliente
        fila.append(último)
    elif operação == "S":
        break
    else:
        print("Operação inválida! Digite apenas F, A ou S!")
```

**Exercício 6.4** O que acontece quando não verificamos se a lista está vazia antes de chamarmos o método pop?

**Exercício 6.5** Altere o Programa 6.7 de forma a poder trabalhar com vários comandos digitados de uma só vez. Atualmente, apenas um comando pode ser inserido por vez. Altere-o de forma a considerar operação como uma string.

Exemplo: FFFAAAS significaria três chegadas de novos clientes, três atendimentos e, finalmente, a saída do programa.

**Exercício 6.6** Modifique o programa para trabalhar com duas filas. Para facilitar seu trabalho, considere o comando A para atendimento da fila 1; e B, para atendimento da fila 2. O mesmo para a chegada de clientes: F para fila 1; e G, para fila 2.

## 6.7 Uso de listas como pilhas

Uma pilha tem uma política de acesso bem definida: novos elementos são adicionados ao topo. A retirada de elementos também é feita pelo topo.

Imagine uma pilha de pratos para lavar. Retiramos o prato que está no topo da pilha para lavar e, se mais alguns pratos chegarem, serão também adicionados ou empilhados ao topo, ou seja, um sobre o outro. Na pilha, o último elemento a chegar é o primeiro a sair (*LIFO – Last In First Out*). Por enquanto, vamos nos concentrar nas pilhas de prato. Veja o Programa 6.8, que simula uma pia de cozinha cheia de pratos.

```
# Programa 6.8 - Pilha de pratos
prato = 5
pilha = list(range(1, prato + 1))
while True:
    print(f"\nExistem {len(pilha)} pratos na pilha")
    print(f"Pilha atual: {pilha}")
    print("Digite E para empilhar um novo prato,")
    print("ou D para desempilhar. S para sair.")
    operação = input("Operação (E, D ou S):")
    if operação == "D":
        if len(pilha) > 0:
            lavado = pilha.pop(-1)
            print(f"Prato {lavado} lavado")
        else:
            print("Pilha vazia! Nada para lavar.")
    elif operação == "E":
        prato += 1  # Novo prato
        pilha.append(prato)
    elif operação == "S":
        break
    else:
        print("Operação inválida! Digite apenas E, D ou S!")
```

Você deve ter percebido que o Programa 6.8 é muito parecido com o Programa 6.7. Isso porque a grande diferença entre pilhas e filas é o elemento que escolhemos para retirar. Em uma fila, o primeiro elemento é retirado primeiro. Já em pilhas, retira-se a partir do último elemento. A única mudança em Python é o valor que passamos para o método `pop`. No caso de uma pilha, como retiramos o último elemento, passamos `-1` a `pop`.

Você pode também observar o uso de pilhas com seu browser de internet. Abra algumas páginas, clicando em seus links. Observe como o histórico de navegação é modificado. Cada novo link adiciona uma página a seu histórico de navegação. Se você clicar em voltar uma página, o browser utilizará a última página que entrou no histórico. Nesse caso, o histórico do browser funciona como uma pilha. Na realidade, ele é um pouco mais complexo, pois permite que voltemos várias vezes e que depois possamos voltar no outro sentido se necessário (avançar).

Agora faça outro teste: volte duas ou três vezes e depois visite um novo link. Dessa vez, o browser deve ter desativado a função de avanço, uma vez que o histórico mudou (você mudou de direção). Tente entender esse processo como duas pilhas, uma à esquerda e outra à direta. Quando você escolhe voltar, desempilhamos um elemento da pilha à esquerda. Ao visitar um novo link, adicionamos um novo elemento a essa mesma pilha. Para simular a opção de avançar, imagine que, ao retirarmos um elemento da pilha à esquerda, devemos adicioná-lo à.da direita. Se escolhermos um endereço novo, apagamos a pilha da direita.

Essas mesmas operações podem ser simuladas com outras estruturas, como uma lista simples e uma variável contendo onde estamos no histórico de navegação. A cada novo endereço, adicionaríamos um novo elemento à lista e atualizaríamos nossa posição. Ao voltarmos, decrementaríamos a posição e a incrementaríamos no caso de avanço. Se mudássemos de direção, bastaria apagar os elementos entre a posição atual e o fim da lista antes de adicionar o novo elemento.

A importância de aprender a manipular listas como filas ou pilhas é entender algoritmos mais complexos no futuro.

**Exercício 6.7** Faça um programa que leia uma expressão com parênteses. Usando pilhas, verifique se os parênteses foram abertos e fechados na ordem correta. Exemplo:

```
(())            OK
()()(()())      OK
())             Erro
```

Você pode adicionar elementos à pilha sempre que encontrar abre parênteses e desempilhá-la a cada fecha parênteses. Ao desempilhar, verifique se o topo da pilha é um abre parênteses. Se a expressão estiver correta, sua pilha estará vazia no final.

## 6.8 Pesquisa

Podemos pesquisar se um elemento está ou não em uma lista, verificando do primeiro ao último elemento se o valor procurado estiver presente:

```
# Programa 6.9 - Pesquisa sequencial
L = [15, 7, 27, 39]
p = int(input("Digite o valor a procurar:"))
achou = False ❶
x = 0
while x < len(L):
    if L[x] == p:
        achou = True ❷
        break ❸
    x += 1
if achou: ❹
    print(f"{p} achado na posição {x}")
else:
    print(f"{p} não encontrado")
```

A pesquisa simplesmente compara todos os elementos da lista com o valor procurado, interrompendo a repetição ao encontrar o primeiro elemento cujo valor é igual ao procurado. É importante saber que podemos ou não encontrar o que procuramos, daí a utilização da variável `achou` em ❶. Essa variável do tipo lógico (booleano) será utilizada para verificar se saímos da repetição por termos achado o que procurávamos ou simplesmente porque visitamos todos os elementos sem encontrar o valor procurado. Veja que `achou` é marcada como **`True`** em ❷, mas dentro de **`if`** com a condição de pesquisa, e antes do **`break`** em ❸. Dessa forma, `achou` só será **`True`** se algum elemento for igual ao valor procurado. Em ❹, verificamos o valor de achou para decidir o que vamos imprimir.

**Exercício 6.8** Modifique o primeiro exemplo (Programa 6.9) de forma a realizar a mesma tarefa, mas sem utilizar a variável `achou`. Dica: observe a condição de saída do **`while`**.

**Exercício 6.9** Modifique o exemplo para pesquisar dois valores. Em vez de apenas `p`, leia outro valor `v` que também será procurado. Na impressão, indique qual dos dois valores foi achado primeiro.

**Exercício 6.10** Modifique o programa do Exercício 6.9 de forma a pesquisar `p` e `v` em toda a lista e informando o usuário a posição onde `p` e a posição onde `v` foram encontrados.

## 6.9 Usando for

Python apresenta uma estrutura de repetição especialmente projetada para percorrer listas. A instrução **for** funciona de forma parecida a **while**, mas a cada repetição utiliza um elemento diferente da lista.

A cada repetição, o próximo elemento da lista é utilizado, o que se repete até o fim da lista. Vamos escrever um programa que utilize **for** para imprimir todos os elementos de uma lista:

```
L = [8, 9, 15]
for e in L: ❶
    print(e) ❷
```

Quando começamos a executar o **for** em ❶, temos e igual ao primeiro elemento da lista, no caso, 8, ou seja, `L[0]`. Em ❷ imprimimos 8, e a execução do programa volta para ❶, em que e passa a valer 9, ou seja, `L[1]`. Na próxima repetição e valerá 15, ou seja, `L[2]`. Depois de imprimir o último número, a repetição é concluída, pois não temos mais elementos a substituir. Se tivéssemos de fazer a mesma tarefa com **while**, teríamos de escrever um programa como:

```
L = [8, 9, 15]
x = 0
while x < len(L):
    e = L[x]
    print(e)
    x += 1
```

Embora a instrução **for** facilite nosso trabalho, ela não substitui completamente **while**. Dependendo do problema, utilizaremos **for** ou **while**. Normalmente utilizaremos **for** quando quisermos processar os elementos de uma lista, um a um. **while** é indicado para repetições nas quais não sabemos ainda quantas vezes vamos repetir ou onde manipulamos os índices de forma não sequencial.

Vale lembrar que a instrução **break** também interrompe o **for**. Vejamos a pesquisa, escrita usando **for**:

```
L = [7, 9, 10, 12]
p = int(input("Digite um número a pesquisar:"))
for e in L:
    if e == p:
        print("Elemento encontrado!")
        break ❶
else: ❷
    print("Elemento não encontrado.")
```

Utilizamos a instrução **break** para interromper a busca depois de encontrarmos o primeiro elemento em ❶. Em ❷ utilizamos um **else**, parecido com o da instrução **if**, para imprimir a mensagem informando que o elemento não foi encontrado. O **else** deve ser escrito na mesma coluna de **for** e só será executado se todos os elementos da lista forem visitados, ou seja, se não utilizarmos a instrução **break**, deixando **for** terminar normalmente.

**Exercício 6.11** Modifique o Programa 6.6 usando **for**. Explique por que nem todos os **while** podem ser transformados em **for**.

NOTA: a instrução **while** também possui uma cláusula opcional **else** que funciona da mesma forma que o **else** do **for**.

## 6.10 Range

Podemos utilizar a função **range** para gerar listas simples. A função **range** não retorna uma lista propriamente dita, mas um gerador ou *generator*. Por enquanto, basta entender como podemos usá-la. Imagine um programa simples que imprime de 0 a 9 na tela:

```
for v in range(10):
    print(v)
```

Execute o programa e veja o resultado. A função **range** gerou números de 0 a 9 porque passamos 10 como parâmetro. Ela normalmente gera valores a partir de 0, logo, ao especificarmos apenas 10, estamos apenas informando onde parar. Experimente mudar de 10 para 20. O programa deve imprimir de 0 a 19. Agora experimente 20000. A vantagem de utilizarmos a função **range** é gerar listas eficientemente, como mostrado no exemplo, sem precisar escrever os 20.000 valores no programa. Outra vantagem de utilizarmos **range** é que apenas um valor é gerado a cada iteração do **for**, evitando que uma lista com 20.000 elementos seja criada na memória do computador.

Com a mesma função **range**, podemos também indicar qual é o primeiro número a gerar. Para isso, utilizaremos dois parâmetros:

```
for v in range(5, 8):
    print(v)
```

Usando 5 como início e 8 como fim, vamos imprimir os números 5, 6 e 7. A notação aqui para o fim é a mesma utilizada com fatias, ou seja, o fim é um intervalo aberto, isto é, não incluso na faixa de valores.

Se acrescentarmos um terceiro parâmetro à função **range**, teremos como saltar entre os valores gerados, por exemplo, `range(0, 10, 2)` gera os pares entre 0 e 10, pois começa de 0 e adiciona 2 a cada elemento. Vejamos um exemplo em que geramos os 10 primeiros múltiplos de 3:

```
for t in range(3, 33, 3):
    print(t, end=" ")
print()
```

Observe que utilizamos uma construção especial com a função **print**, em que t é o valor que queremos imprimir, mas com `end=" "`, que indica a função para não pular de linha após a impressão. `end` é, na realidade, um parâmetro opcional da função **print**. Veremos mais sobre isso quando estudarmos funções no Capítulo 8. Veja também que, para saltar a linha no final do programa, fizemos uma chamada a `print()` sem qualquer parâmetro.

Observe que um gerador como o retornado pela função **range** não é exatamente uma lista. Embora seja usado de forma parecida, é, na realidade, um objeto de outro tipo. Para transformar um gerador em lista, utilize a função **list**:

```
# Programa 6.10 - Transformação de range em uma lista
L = list(range(100, 1100, 50))
print(L)
```

## 6.11 Enumerate

Com a função **enumerate** podemos ampliar as funcionalidades de **for** facilmente. Vejamos como imprimir uma lista, em que teremos o índice entre colchetes e o valor à sua direita:

```
L = [5, 9, 13]
x = 0
for e in L:
    print(f"[{x}] {e}")
    x += 1
```

Veja o mesmo programa, mas utilizando a função **enumerate**:

```
L = [5, 9, 13]
for x, e in enumerate(L):
    print(f"[{x}] {e}")
```

A função **enumerate** gera uma tupla em que o primeiro valor é o índice e o segundo é o elemento da lista sendo enumerada. Ao utilizarmos `x, e` em **for**, indicamos que o primeiro valor da tupla deve ser colocado em `x`, e o segundo, em `e`. Assim, na primeira iteração teremos a tupla (0, 5), em que `x = 0` e `e = 5`. Isso é possível porque Python permite o desempacotamento de valores de uma tupla, atribuindo um elemento da tupla a cada variável em **for**. O que temos a cada iteração de **for** é equivalente a `x, e = (0, 5)`, em que o gerador **enumerate** retorna cada vez uma nova tupla. Os próximos valores retornados são (1, 9) e (2, 13), respectivamente. Experimente substituir `x, e` no Programa 6.10 por `z`. Antes de **print**, faça `x, e = z`. Adicione mais um **print** para exibir também o valor de `z`. Veremos mais sobre tuplas na Seção 6.20.

## 6.12 Operações com listas

Podemos percorrer uma lista de forma a verificar o menor e o maior valor:

```
# Programa 6.11 - Verificação do maior valor
L = [1, 7, 2, 4]
máximo = L[0] ❶
for e in L:
    if e > máximo:
        máximo = e
print(máximo)
```

Em ❶, utilizamos um pequeno truque, inicializando o máximo com o valor do primeiro elemento. Precisamos de um valor para máximo antes de utilizá-lo na comparação com **if**. Se usássemos 0, não teríamos problema, desde que nossa lista não tenha valores negativos.

**Exercício 6.12** Altere o Programa 6.11 de forma a imprimir o menor elemento da lista.

**Exercício 6.13** A lista de temperaturas de Mons, na Bélgica, foi armazenada na lista `T = [ -10, -8, 0, 1, 2, 5, -2, -4]`. Faça um programa que imprima a menor e a maior temperatura, assim como a temperatura média.

## 6.13 Aplicações

Vejamos uma situação na qual temos de selecionar os elementos de uma lista de forma a copiá-los para outras duas listas. Para simplificar o problema, imagine que os valores estejam inicialmente na lista `V`, mas que devam ser copiados para a lista `P`, se forem pares; ou para a lista `I`, se forem ímpares. Veja o programa que resolve esse problema:

```
# Programa 6.12 - Cópia de elementos para outras listas
V = [9, 8, 7, 12, 0, 13, 21]
P = []
I = []
for e in V:
    if e % 2 == 0:
        P.append(e)
    else:
        I.append(e)
print("Pares: ", P)
print("Impares: ", I)
```

Vejamos agora um programa que controla a utilização das salas de um cinema. Imagine que a lista `lugares_vagos = [10, 2, 1, 3, 0]` contenha o número de lugares vagos nas salas 1, 2, 3, 4 e 5, respectivamente. Esse programa lerá o número da sala e a quantidade de lugares solicitados. Ele deve informar se é possível vender o número de lugares solicitados, ou seja, se ainda há lugares livres. Caso seja possível vender os bilhetes, atualizará o número de lugares livres. A saída ocorre quando se digita 0 no número da sala.

```
# Programa 6.13 - Controle da utilização de salas de um cinema
lugares_vagos = [10, 2, 1, 3, 0]
while True:
    sala = int(input("Sala (0 sai): "))
    if sala == 0:
        print("Fim")
        break
    if sala > len(lugares_vagos) or sala < 1:
        print("Sala inválida")
    elif lugares_vagos[sala - 1] == 0:
        print("Desculpe, sala lotada!")
    else:
        lugares = int(input(f"Quantos lugares você deseja ({lugares_vagos[sala - 1]}
    vagos):"))
```

```
        if lugares > lugares_vagos[sala - 1]:
            print("Esse número de lugares não está disponível.")
        elif lugares < 0:
            print("Número inválido")
        else:
            lugares_vagos[sala - 1] -= lugares
            print(f"{lugares} lugares vendidos")
print("Utilização das salas")
for x, l in enumerate(lugares_vagos):
    print(f"Sala {x + 1} - {l} lugar(es) vazio(s)")
```

## 6.14 Listas com strings

No Capítulo 3, vimos que strings podem ser indexadas ou acessadas letra por letra. Listas em Python funcionam da mesma forma, permitindo o acesso a vários valores e se tornando uma das principais estruturas de programação da linguagem.

Vejamos agora outro exemplo de listas, mas utilizando strings. Nesse caso, `S` é uma lista, e cada elemento, uma string:

```
>>> S = ["maçãs", "peras", "kiwis"]
>>> len(S)
3
>>> S[0]
maçãs
>>> S[1]
peras
>>> S[2]
kiwis
```

Vejamos um programa que lê e imprime uma lista de compras até que seja digitado fim.

```
# Programa 6.14 - Lendo e imprimindo uma lista de compras
compras = []
while True:
    produto = input("Produto:")
    if produto == "fim":
        break
    compras.append(produto)
for p in compras:
    print(p)
```

## 6.15 Listas dentro de listas

Um fator interessante é que podemos acessar as strings dentro da lista, letra por letra, usando um segundo índice:

```
>>> S = ["maçãs", "peras", "kiwis"]
>>> print(S[0][0])
m
>>> print(S[0][1])
a
>>> print(S[1][1])
e
>>> print(S[2][2])
w
```

Ou utilizando um programa:

```
# Programa 6.15 - Impressão de uma lista de strings, letra a letra
L = ["maçãs", "peras", "kiwis"]
for s in L:
    for letra in s:
        print(letra)
```

Isso nos leva a outra vantagem das listas em Python: listas dentro de listas. Como bônus, temos também que os elementos de uma lista não precisam ser do mesmo tipo. Vejamos um exemplo em que teríamos uma lista de compras. O primeiro elemento seria o nome do produto; o segundo, a quantidade; e o terceiro, seu preço.

```
# Programa 6.16 - Listas com elementos de tipos diferentes
produto1 = ["maçã", 10, 0.30]
produto2 = ["pera",  5, 0.75]
produto3 = ["kiwi",  4, 0.98]
```

Assim, `produto1`, `produto2`, `produto3` seriam três listas com três elementos cada uma. Observe que o primeiro elemento é do tipo string; o segundo, do tipo inteiro; e o terceiro, do tipo ponto flutuante (*float*)!

Vejamos agora outra lista:

```
# Programa 6.17 - Listas de listas
produto1 = ["maçã", 10, 0.30]
produto2 = ["pera",  5, 0.75]
produto3 = ["kiwi",  4, 0.98]
compras = [produto1, produto2, produto3]
print(compras)
```

Agora temos uma lista chamada compras, também com três elementos, mas cada elemento é uma lista à parte. Para imprimir essa lista, teríamos o programa:

```
# Programa 6.18 - Impressão das compras
produto1 = ["maçã", 10, 0.30]
produto2 = ["pera",  5, 0.75]
produto3 = ["kiwi",  4, 0.98]
compras = [produto1, produto2, produto3]
for e in compras:
    print(f"Produto: {e[0]}")
    print(f"Quantidade: {e[1]}")
    print(f"Preço: {e[2]:5.2f}")
```

Da mesma forma, poderíamos ter acessado o preço do segundo elemento com compras[1][2]. Vejamos agora um programa completo, capaz de perguntar nome do produto, quantidade e preço e, no final, imprimir uma lista de compras completa.

```
# Programa 6.19 - Criação e impressão da lista de compras
compras = []
while True:
    produto = input("Produto: ")
    if produto == "fim":
        break
    quantidade = int(input("Quantidade: "))
    preço = float(input("Preço: "))
    compras.append([produto, quantidade, preço])
soma = 0.0
for e in compras:
    print(f"{e[0]:20s} x {e[1]:5d} {e[2]:5.2f} {e[1] * e[2]:6.2f}")
    soma += e[1] * e[2]
print(f"Total: {soma:7.2f}")
```

## 6.16 Ordenação

Até agora, os elementos de nossas listas apresentam a mesma ordem em que foram digitados, sem qualquer ordenação. Para ordenar uma lista, realizaremos uma operação semelhante à da pesquisa, mas trocando a ordem dos elementos quando necessário. Um algoritmo muito simples de ordenação é o *Bubble Sort*, ou método de bolhas, fácil de entender e aprender. Por ser lento, você não deve utilizá-lo com listas grandes.

A ordenação pelo método de bolhas consiste em comparar dois elementos a cada vez. Se o valor do primeiro elemento for maior que o do segundo, eles trocarão de posição. Essa operação é então repetida para o próximo elemento até o fim da lista. O método de bolhas exige que percorramos a lista várias vezes. Por isso, utilizaremos um marcador para saber se chegamos ao fim da lista trocando ou não algum elemento. Esse método tem outra propriedade, que é posicionar o maior elemento na última posição da lista, ou seja, sua posição correta. Isso permite eliminar um elemento do fim da lista a cada passagem completa. Vejamos o programa:

```
# Programa 6.20 - Ordenação pelo método de bolhas
L = [7, 4, 3, 12, 8]
fim = 5 ❶
while fim > 1: ❷
    trocou = False ❸
    x = 0 ❹
    while x < (fim - 1): ❺
        if L[x] > L[x + 1]: ❻
            trocou = True ❼
            temp = L[x] ❽
            L[x] = L[x + 1]
            L[x + 1] = temp
        x += 1
    if not trocou: ❾
        break
    fim -= 1 ❿
for e in L:
    print(e)
```

Execute o programa e tente entender como ele funciona. Em ❶, utilizamos a variável `fim` para marcar a quantidade de elementos da lista. Precisamos marcar o fim ou a posição do último elemento porque no *Bubble Sort* não precisamos verificar o último elemento após uma passagem completa. Em ❷, verificamos se `fim > 1`, pois, como comparamos o elemento atual(`L[x]`) com o seguinte (`L[x + 1]`), precisamos de, no mínimo, dois elementos. Utilizamos a variável `trocou` em ❸ para indicar que não realizamos nenhuma troca. O valor de `trocou` será utilizado mais tarde para verificar se já temos a lista ordenada ou não. A variável `x` ❹ será utilizada como índice, começando da posição 0. Nossa condição do segundo **`while`** ❺ é especial, pois temos de garantir um próximo elemento para comparar

com o atual. Isso faz com que a condição de saída desse `while` seja `x < (fim - 1)`, ou melhor, que `x` seja anterior ao último elemento. Em ❻ temos a comparação em si, em que `x` é nossa posição atual, e o próximo elemento é o de índice `x + 1`. Como estamos ordenando em ordem crescente, desejamos que o próximo elemento sempre seja maior que o atual, dessa forma garantindo a ordenação da lista. Como comparamos apenas dois elementos de cada vez, temos de visitar a lista várias vezes, até que todos os elementos estejam em suas posições. Se a condição de ❻ for verdadeira, esses elementos estão fora de ordem. Nesse caso, devemos inverter ou trocar a posição dos dois elementos. Em ❼ marcamos que efetuamos uma troca e, em ❽, utilizamos uma variável auxiliar ou temporária para trocar os dois valores de posição.

A troca de valores entre duas variáveis é mostrada nas figuras 6.3, 6.4 e 6.5. Como cada variável só guarda um valor de cada vez, utilizaremos uma terceira variável temporária (`temp`) para armazenar o valor de uma delas durante a troca.

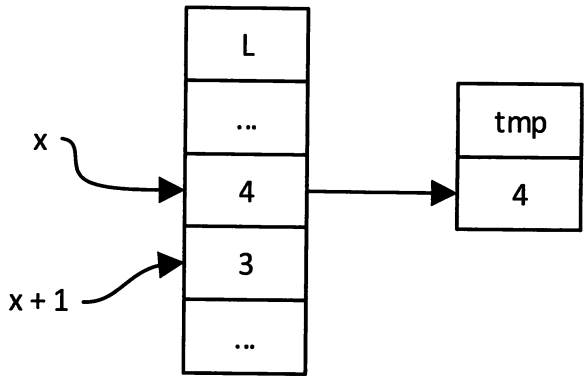


*Figura 6.3 – Troca passo a passo. Primeira etapa.*

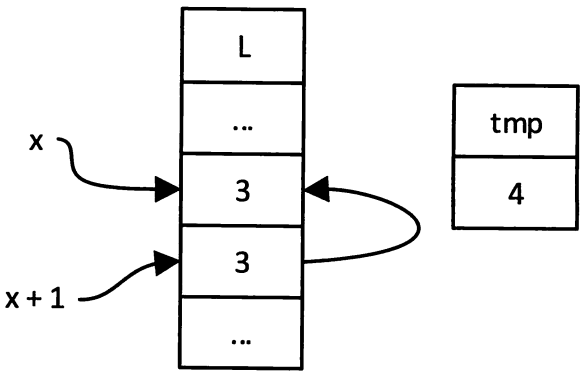


*Figura 6.4 – Troca passo a passo. Segunda etapa.*

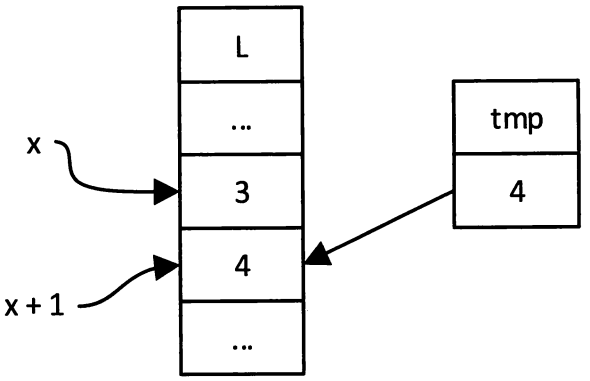


*Figura 6.5 – Troca passo a passo. Terceira etapa.*

Você pode entender melhor o processo de troca de valores utilizando um problema do dia a dia: imagine que você tenha duas xícaras, uma com leite e outra com café. O problema consiste em passar o leite para a xícara que contém o café, e o café para a xícara que contém o leite, ou seja, você deve trocar o conteúdo das xícaras. Nesse caso, resolveríamos o problema utilizando uma terceira xícara para facilitar a operação. Fizemos o mesmo durante a ordenação, em que chamamos nossa terceira variável de `temp`.

Quando a repetição iniciada em ❺ termina, o maior elemento está posicionado na última posição da lista, ou seja, em sua posição correta. Em ❾, verificamos se algo foi trocado na repetição anterior. Se não trocamos nada de lugar, nossa lista está ordenada, e não precisamos executar a repetição outra vez, por isso o **break**. Caso contrário, como a última posição já está com o elemento correto, diminuiremos o valor de `fim` para que não precisemos mais verificá-lo.

Vamos rastrear o algoritmo e observar como tudo isso funciona. Observe o rastreamento completo na Tabela 6.1. A coluna linha indica a linha do programa sendo executada: não confunda com os números dentro dos círculos brancos da listagem.

**Exercício 6.14** O que acontece quando a lista já está ordenada? Rastreie o Programa 6.20, mas com a lista `L = [1, 2, 3, 4, 5]`.

**Exercício 6.15** O que acontece quando dois valores são iguais? Rastreie o Programa 6.20, mas com a lista `L = [3, 3, 1, 5, 4]`.

**Exercício 6.16** Modifique o Programa 6.20 para ordenar a lista em ordem decrescente. `L = [1, 2, 3, 4, 5]` deve ser ordenada como `L = [5, 4, 3, 2, 1]`.

*Tabela 6.1 – Rastreamento da ordenação com Bubble Sort*

| Linha | L [0] | L[1] | L [2] | L [3] | L[4] | fim | trocou | x | temp |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 7 | 4 | 3 | 12 | 8 | | | | |
| 2 | | | | | | 5 | | | |
| 4 | | | | | | | False | | |
| 5 | | | | | | | | 0 | |
| 8 | | | | | | | True | | |
| 9 | | | | | | | | | 7 |
| 10 | 4 | | | | | | | | |
| 11 | | 7 | | | | | | | |
| 12 | | | | | | | | 1 | |
| 8 | | | | | | | True | | |
| 9 | | | | | | | | | 7 |
| 10 | | 3 | | | | | | | |
| 11 | | | 7 | | | | | | |
| 12 | | | | | | | | 2 | |
| 12 | | | | | | | | 3 | |
| 8 | | | | | | | True | | |
| 9 | | | | | | | | | 12 |
| 10 | | | | 8 | | | | | |
| 11 | | | | | 12 | | | | |
| 12 | | | | | | | | 4 | |
| 15 | | | | | | 4 | | | |
| 4 | | | | | | | False | | |
| 5 | | | | | | | | 0 | |
| 8 | | | | | | | True | | |
| 9 | | | | | | | | | 4 |
| 10 | 3 | | | | | | | | |
| 11 | | 4 | | | | | | | |
| 12 | | | | | | | | 1 | |
| 12 | | | | | | | | 2 | |
| 12 | | | | | | | | 3 | |
| 15 | | | | | | 3 | | | |
| 4 | | | | | | | False | | |
| 5 | | | | | | | | 0 | |
| 12 | | | | | | | | 1 | |
| 12 | | | | | | | | 2 | |

NOTA: Python possui o método sort que ordena rapidamente uma lista.

```
>>> L = [6, 4, 2, 1, 9]
>>> L.sort()
>>> L
[1, 2, 4, 6, 9]
```

Se você deseja ordenar uma lista, sem alterar seus elementos, utilize a função **sorted**, que retorna a nova lista ordenada:

```
>>> Z = [4, 3, 1]
>>> sorted(Z)
[1, 3, 4]
>>> Z
[4, 3, 1]
```

**Trivia**

Existem vários algoritmos de ordenação. O método sort do Python utiliza o método chamado TimSort (https://pt.wikipedia.org/wiki/Timsort). Outros algoritmos interessantes de aprender são o Merge Sort (https://pt.wikipedia.org/wiki/Merge_sort), Quick Sort (https://pt.wikipedia.org/wiki/Quicksort) e Insertion Sort (https://pt.wikipedia.org/wiki/Insertion_sort). Fale com seu professor de estrutura de dados ou algoritmos para saber mais sobre os diferentes métodos de ordenação (https://pt.wikipedia.org/wiki/Algoritmo_de_ordena%C3%A7%C3%A3o).

## 6.17 Dicionários

Dicionários consistem em uma estrutura de dados similar às listas, mas com propriedades de acesso diferentes. Um dicionário é composto de um conjunto de chaves e valores. O dicionário em si consiste em relacionar uma chave a um valor específico.

Em Python, criamos dicionários utilizando chaves ({}). Cada elemento do dicionário é uma combinação de chave e valor. Vejamos um exemplo em que os preços de mercadorias sejam como os da Tabela 6.2.

*Tabela 6.2 – Preços de mercadorias*

| Produto | Preço |
|---|---|
| Alface | R$ 0,45 |
| Batata | R$ 1,20 |

| Produto | Preço |
|---|---|
| Tomate | R$ 2,30 |
| Feijão | R$ 1,50 |

A Tabela 6.2 pode ser vista como um dicionário, em que chave seria o produto; e valor, seu preço. Vejamos como criar esse dicionário em Python:

```
tabela = {"Alface": 0.45,
          "Batata": 1.20,
          "Tomate": 2.30,
          "Feijão": 1.50}
```

Um dicionário é acessado por suas chaves. Para obter o preço da alface, digite no interpretador, depois de ter criado a tabela, `tabela["Alface"]`, em que `tabela` é o nome da variável do tipo dicionário, e "Alface" é nossa chave. O valor retornado é o mesmo que associamos na tabela, ou seja, 0.45.

Diferentemente de listas, em que o índice é um número, dicionários utilizam suas chaves como índice. Quando atribuímos um valor a uma chave, duas coisas podem ocorrer:

1. Se a chave já existe: o valor associado é alterado para o novo valor.
2. Se a chave não existe: a nova chave será adicionada ao dicionário.

Observe:

```
>>> tabela = {"Alface": 0.45,
...           "Batata": 1.20,
...           "Tomate": 2.30,
...           "Feijão": 1.50}
>>> print(tabela["Tomate"]) ❶
2.3
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Tomate': 2.3, 'Feijão': 1.5}
>>> tabela["Tomate"] = 2.50 ❷
>>> print(tabela["Tomate"])
2.5
>>> tabela["Cebola"] = 1.20 ❸
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Tomate': 2.5, 'Cebola': 1.2, 'Feijão': 1.5}
```

Em ❶, acessamos o valor associado à chave "Tomate". Em ❷, alteramos o valor associado à chave "Tomate" para um novo valor. Observe que o valor anterior

foi perdido. Em ❸, criamos uma nova chave, "Cebola", que é adicionada ao dicionário. Veja também como Python imprime o dicionário. Outra diferença entre dicionários e listas é que, ao utilizarmos dicionários, perdemos a noção de ordem. Observe que, durante a manipulação do dicionário, a ordem das chaves foi alterada (a partir do Python 3.7, a ordem de inserção de elementos em um dicionário é mantida).

Quanto ao acesso aos dados, temos de verificar se uma chave existe, antes de acessá-la:

```
>>> tabela = {"Alface": 0.45,
...           "Batata": 1.20,
...           "Tomate": 2.30,
...           "Feijão": 1.50}
>>> print(tabela["Manga"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Manga'
```

Se a chave não existir, uma exceção do tipo `KeyError` será ativada. Para verificar se uma chave pertence ao dicionário, podemos usar o operador **`in`**:

```
>>> tabela = {"Alface": 0.45,
...           "Batata": 1.20,
...           "Tomate": 2.30,
...           "Feijão": 1.50}
>>> print("Manga" in tabela)
False
>>> print("Batata" in tabela)
True
```

Podemos também obter uma lista com as chaves do dicionário, ou mesmo uma lista dos valores associados:

```
>>> tabela = {"Alface": 0.45,
...           "Batata": 1.20,
...           "Tomate": 2.30,
...           "Feijão": 1.50}
>>> print(tabela.keys())
dict_keys(['Batata', 'Alface', 'Tomate', 'Feijão'])
>>> print(tabela.values())
dict_values([1.2, 0.45, 2.3, 1.5])
```

Observe que os métodos `keys()` e `values()` retornam geradores. Você pode utilizá-los diretamente dentro de um **`for`** ou transformá-los em lista usando a função **`list`**.

Vejamos um programa que utiliza dicionários para exibir o preço de um produto:

```
# Programa 6.21 - Obtenção do preço com um dicionário
tabela = {"Alface": 0.45,
          "Batata": 1.20,
          "Tomate": 2.30,
          "Feijão": 1.50}
while True:
    produto = input("Digite o nome do produto, fim para terminar:")
    if produto == "fim":
        break
    if produto in tabela: ❶
        print(f"Preço {tabela[produto]:5.2f}") ❷
    else:
        print("Produto não encontrado!")
```

Em ❶ verificamos se o dicionário contém a chave procurada. Em caso afirmativo, imprimimos o preço associado à chave, ou haverá uma mensagem de erro.

Para apagar uma chave, utilizaremos a instrução **del**:

```
>>> tabela = {"Alface": 0.45,
...           "Batata": 1.20,
...           "Tomate": 2.30,
...           "Feijão": 1.50}
>>> del tabela["Tomate"]
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Feijão': 1.5}
```

Você pode estar se perguntando quando utilizar listas e quando utilizar dicionários. Tudo depende do que você deseja realizar. Se seus dados são facilmente acessados por suas chaves e quase nunca você precisa acessá-los de uma só vez: um dicionário é mais interessante. Além disso, você pode acessar os valores associados a uma chave rapidamente sem pesquisar. A implementação interna de dicionários também garante uma boa velocidade de acesso quando temos muitas chaves. Porém, um dicionário não organiza suas chaves, ou seja, as primeiras chaves inseridas nem sempre serão as primeiras na lista de chaves (salvo a partir da versão 3.7 do Python, na qual a ordem de inserção é mantida). Se seus dados precisam preservar a ordem de inserção (como em filas ou pilhas, continue a usar listas), dicionários não serão uma opção.

NOTA: Atenção ao usar f-strings com dicionários:

```
>>> d = {"banana": 2.00, "maçã": 5.00}
>>> f"Banana: R${d['banana']:5.2f} Maçã: R${d['maçã']:5.2f}"
'Banana: R$ 2.00 Maçã: R$ 5.00'
```

Veja que " foram substituídas por '. Você também pode evitar esse problema usando """.

**Trivia**

Dicionários usam uma técnica chamada espalhamento ou hash (*). Essa técnica utiliza um algoritmo que calcula um número ou hash para a chave que estamos procurando. Esse valor é sempre igual se o valor da chave também for igual. Dessa forma, ao procurarmos uma chave, o seu hash é calculado e utilizado para procurar o índice desejado. A técnica é mais complexa que isso, pois precisa lidar com colisões (chaves diferentes que têm o mesmo hash). Veja mais informações na página da Wikipédia sobre o assunto ou durante um curso de estrutura de dados.

Além de strings, os dicionários em Python aceitam outros tipos de dados, entre eles: números e tuplas. Python tem uma função padrão, chamada **hash**, que é usada para calcular o número hash de chaves.

Uma demonstração do uso da função **hash**:

```
>>> L = [0] * 10
>>> hash("A")
6082593157453147583
>>> hash("A") % 10  # Se utilizarmos o resto da divisão entre o hash e o tamanho da
   lista, teremos um índice a partir da chave
3
>>> L[hash("A") % 10] = "A"
>>> L
[0, 0, 0, 'A', 0, 0, 0, 0, 0, 0]
```

(*) https://pt.wikipedia.org/wiki/Tabela_de_dispers%C3%A3o

## 6.18 Dicionários com listas

Em Python, podemos ter dicionários nos quais as chaves são associadas a listas ou mesmo a outros dicionários. Imagine uma relação de estoque de mercadorias na qual teríamos, além do preço, a quantidade em estoque:

```
estoque = {"tomate": [1000, 2.30],
           "alface": [ 500, 0.45],
           "batata": [2001, 1.20],
           "feijão": [ 100, 1.50]}
```

Nesse caso, o nome do produto é a chave, e a lista consiste nos valores associados, uma lista por chave. O primeiro elemento da lista é a quantidade disponível; e o segundo, o preço do produto.

Uma aplicação seria processarmos uma lista de operações e calcular o preço total de venda, atualizando também a quantidade em estoque.

```
# Programa 6.22 - Exemplo de dicionário com estoque e operações de venda
estoque = {"tomate": [1000, 2.30],
           "alface": [ 500, 0.45],
           "batata": [2001, 1.20],
           "feijão": [ 100, 1.50]}
venda = [["tomate", 5], ["batata", 10], ["alface",5]]
total = 0
print("Vendas:\n")
for operação in venda:
    produto, quantidade = operação ❶
    preço = estoque[produto][1] ❷
    custo = preço * quantidade
    print(f"{produto:12s}: {quantidade:3d} x {preço:6.2f} = {custo:6.2f}")
    estoque[produto][0] -= quantidade ❸
    total += custo
print(f" Custo total: {total:21.2f}\n")
print("Estoque:\n")
for chave, dados in estoque.items(): ❹
    print("Descrição: ", chave)
    print("Quantidade: ", dados[0])
    print(f"Preço: {dados[1]:6.2f}\n")
```

Em ❶ utilizamos uma operação de desempacotamento, como já fizemos com **for** e **enumerate**. Como `operação` é uma lista com dois elementos, ao escrevermos `produto, quantidade` temos o primeiro elemento de `operação` atribuído a `produto`; e o segundo, a `quantidade`. Dessa forma, a construção é equivalente a:

```
produto = operação[0]
quantidade = operação[1]
```

Em ❷, utilizamos o conteúdo de `produto` como chave no dicionário `estoque`. Como nossos dados são uma lista, escolhemos o segundo elemento, que armazena o

preço do referido produto. Observe que atribuir nomes a cada um desses componentes facilita a leitura do programa.

Imagine escrever:

```
preço = estoque[operação[0]][1]
```

Em ❸, atualizamos a quantidade em estoque subtraindo a quantidade vendida do estoque atual.

Já em ❹ utilizamos o método `items` do objeto dicionário. O método `items` é um gerador que retorna uma tupla contendo a chave e o valor de cada item armazenado no dicionário. Usando um **`for`** com duas variáveis, `chave` e `dados`, efetuamos o desempacotamento desses valores em uma só passagem. Para entender melhor como isso acontece, experimente alterar o programa para exibir o valor de chave e dados a cada iteração.

**Exercício 6.17** Altere o Programa 6.22 de forma a solicitar ao usuário o produto e a quantidade vendida. Verifique se o nome do produto digitado existe no dicionário, e só então efetue a baixa em estoque.

**Exercício 6.18** Escreva um programa que gere um dicionário, em que cada chave seja um caractere, e seu valor seja o número desse caractere encontrado em uma frase lida.

Exemplo: O rato -> { "O":1, "r":1, "a":1, "t":1, "o":1}

## 6.19 Dicionários com valor padrão

Uma operação muito comum com dicionários é a de recuperar o valor de uma chave e, caso esta não exista, utilizar um valor padrão. Vejamos um exemplo em que utilizaremos um dicionário para contar o número de ocorrências de uma letra em uma string:

```
# Programa 6.23 - Exemplo de dicionário sem valor padrão
d = {}
for letra in "abracadabra":
    if letra in d:
        d[letra] = d[letra] + 1
    else:
        d[letra] = 1
print(d)
```

Que imprime ao ser executado:

```
{'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}
```

Porém, utilizando o método get do dicionário, poderíamos simplificar o código para:

```
# Program 6.24 - Exemplo de dicionário com valor padrão
d = {}
for letra in "abracadabra":
    d[letra] = d.get(letra, 0) + 1
print(d)
```

O método get tenta obter a chave procurada; caso não a encontre, retorna o segundo parâmetro, no caso 0 (zero). Se o segundo parâmetro não for especificado, get retornará **None**. Quando a chave é encontrada no dicionário, get retorna o valor atualmente associado.

**None** é um tipo especial no Python que representa nada ou nenhum valor. Assim, uma string vazia "" não é a mesma coisa que **None**. Todo objeto Python pode receber **None**.

## 6.20 Tuplas

Tuplas podem ser vistas como listas em Python, com a grande diferença de serem imutáveis. Tuplas são ideais para representar listas de valores constantes e para realizar operações de empacotamento e desempacotamento de valores. Primeiro, vejamos como criar uma tupla.

Tuplas são criadas de forma semelhante às listas, mas utilizamos parênteses em vez de colchetes. Por exemplo:

```
>>> tupla = ("a", "b", "c")
>>> tupla
('a', 'b', 'c')
```

Os parênteses são opcionais, porém sua utilização aumenta a clareza da expressão em si, uma vez que visualizar a vírgula pode não ser tão fácil. Você poderia criar a mesma tupla usando a sintaxe seguinte:

```
>>> tupla = "a", "b", "c"
>>> tupla
('a', 'b', 'c')
```

Tuplas suportam a maior parte das operações de lista, como fatiamento e indexação:

```
>>> tupla[0]
'a'
>>> tupla[2]
'c'
>>> tupla[1:]
('b', 'c')
>>> tupla * 2
('a', 'b', 'c', 'a', 'b', 'c')
>>> len(tupla)
3
```

Mas tuplas não podem ter seus elementos alterados. Veja o que acontece se tentarmos alterar uma tupla:

```
>>> tupla[0] = "A"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Várias funções utilizam ou geram tuplas em Python. Tuplas podem ser utilizadas com **for**:

```
>>> for elemento in tupla:
...     print(elemento)
...
a
b
c
```

Vejamos outro exemplo de tupla:

```
>>> tupla = 100, 200, 300
>>> tupla
(100, 200, 300)
```

No caso, 100, 200 e 300 foram convertidos em uma tupla com três elementos. Esse tipo de operação é chamado de empacotamento.

Tuplas também podem ser utilizadas para desempacotar valores, por exemplo:

```
>>> a, b = 10, 20
>>> a
10
>>> b
20
```

Em que o primeiro valor, 10, foi atribuído à primeira variável `a` e 20, à segunda, `b`.

Esse tipo de construção é interessante para distribuirmos o valor de uma tupla em várias variáveis.

Também podemos trocar rapidamente os valores de variáveis com construções do tipo:

```
>>> a, b = 10, 20
>>> a, b = b, a
>>> a
20
>>> b
10
```

Em que a tupla da esquerda foi usada para atribuir os valores à direita. Nesse caso, as atribuições: `a = b` e `b = a` foram realizadas imediatamente, sem precisar utilizarmos uma variável intermediária para a troca.

A sintaxe do Python é um tanto especial quando precisamos criar tuplas com apenas um elemento. Como os valores são escritos entre parênteses, quando apenas um valor estiver presente, devemos acrescentar uma vírgula para indicar que o valor é uma tupla com apenas um elemento. Veja o que acontece, usando e não usando a vírgula:

```
>>> t1 = (1)
>>> t1
1
>>> t2 = (1,)
>>> t2
(1,)
>>> t3 = 1,
>>> t3
(1,)
```

Veja que em `t1` não utilizamos a vírgula, e o código foi interpretado como um número inteiro entre parênteses. Já em `t2`, utilizamos a vírgula, e nossa tupla foi corretamente construída. Em `t3`, criamos outra tupla, mas nesse caso nem precisamos usar parênteses.

Podemos também criar tuplas vazias, escrevendo apenas os parênteses:

```
>>> t4 = ()
>>> t4
()
>>> len(t4)
0
```

Tuplas também podem ser criadas a partir de listas, utilizando-se a função **tuple**:

```
>>> L = [1, 2, 3]
>>> T = tuple(L)
>>> T
(1, 2, 3)
```

Embora não possamos alterar uma tupla depois de sua criação, podemos concatená-las, gerando novas tuplas:

```
>>> t1 = (1, 2, 3)
>>> t2 = (4, 5, 6)
>>> t1 + t2
(1, 2, 3, 4, 5, 6)
```

Observe que, se uma tupla contiver uma lista ou outro objeto que pode ser alterado, este continuará a funcionar normalmente. Veja o exemplo de uma tupla que contém uma lista:

```
>>> tupla = ("a", ["b", "c", "d"])
>>> tupla
('a', ['b', 'c', 'd'])
>>> len(tupla)
2
>>> tupla[1]
['b', 'c', 'd']
>>> tupla[1].append("e")
>>> tupla
('a', ['b', 'c', 'd', 'e'])
```

Nesse caso, nada mudou na tupla em si, mas na lista que é seu segundo elemento. Ou seja, a tupla não foi alterada, mas a lista que ela continha sim.

As operações de empacotamento e desempacotamento também funcionam com listas.

```
>>> a, b = 1, 2
>>> a
1
>>> b
2
>>> b, c = [3, 4]
>>> b
3
>>> c
4
```

Podemos usar o * para indicar vários valores a desempacotar.

```
>>> *a, b = [1, 2, 3, 4, 5]
>>> a
[1, 2, 3, 4]
>>> b
5
>>> a, *b = [1, 2, 3, 4, 5]
>>> a
1
>>> b
[2, 3, 4, 5]
```

No caso de `*a, b`, dizemos: coloque o último valor em `b` e os restantes em `a`. Em `a, *b`, dizemos: coloque o primeiro valor em `a` e os outros em `b`.

Você pode entender essa sintaxe de forma que a variável sem o * receberá apenas um valor, no caso, o valor na mesma posição da variável. Já a variável com * receberá os valores que sobraram após a distribuição dos valores às variáveis sem o *. Em `*c, d, e`, podemos dizer que `c` é a variável com o *, logo receberá todos os valores, salvo aqueles alocados a `d` e `e` (penúltimo e último valor, respectivamente). Vejamos mais alguns exemplos. Não hesite em experimentar com o interpretador.

```
>>> a, *b, c = [1, 2, 3, 4, 5]
>>> a
1
>>> b
[2, 3, 4]
>>> c
5
>>> *a, b, c = [1, 2, 3, 4, 5]
>>> a
[1, 2, 3]
>>> b
4
>>> c
5
>>> a, b, *c = [1, 2, 3, 4, 5]
>>> a
1
>>> b
2
>>> c
[3, 4, 5]
```

## 6.21 Conjuntos (set)

Conjuntos, **set** em Python, são uma estrutura de dados que implementam operações de união, intersecção e diferença, entre outras. A principal característica dessa estrutura de dados é não admitir repetição de elementos, como os conjuntos da matemática. Outra característica importante é que conjuntos não mantêm a ordem de seus elementos. Vejamos algumas operações com conjuntos.

```
>>> a = set()
>>> a.add(1)
>>> a.add(2)
>>> a.add(3)
>>> a
{1, 2, 3}
>>> a.add(1)
>>> a
{1, 2, 3}
```

No exemplo, veja que, ao adicionarmos 1 pela segunda vez, nada alteramos no conjunto em si, uma vez que 1 já fazia parte deste. A ordem relativa que vemos no exemplo é mera coincidência.

```
>>> a.add(0)
>>> a.add(-1)
{0, 1, 2, 3, -1}
```

Podemos testar se um elemento faz parte de um conjunto usando o operador **in** do Python:

```
>>> 1 in a
True
>>> -2 in a
False
```

Um **set**(conjunto) pode ser criado a partir de listas, tuplas e qualquer outra estrutura de dados que seja enumerável.

```
>>> b = set([2, 3])
>>> b
{2, 3}
```

Entre as operações disponíveis com conjuntos, temos a diferença entre conjuntos, que utiliza o operador -.

```
>>> a = {0, 1, 2, 3, -1}
>>> b = {2, 3}
>>> a - b
{0, 1, -1}
```

O resultado contém apenas os elementos de a que não estão presentes em b.

A união é realizada pelo operador |:

```
>>> a = {0, 1, 2, 3, -1}
>>> b = {2, 3}
>>> c = set([4, 5, 6])
>>> a | b
{0, 1, 2, 3, -1}
>>> a | c
{0, 1, 2, 3, 4, 5, 6, -1}
```

Conjuntos (**set**) também possuem outras propriedades, como tamanho (número de elementos), que podem ser obtidas com o uso da função **len**:

```
>>> len(a)
5
>>> len(c)
3
```

**Exercício 6.19** Escreva um programa que compare duas listas. Utilizando operações com conjuntos, imprima:

- os valores comuns às duas listas
- os valores que só existem na primeira
- os valores que existem apenas na segunda
- uma lista com os elementos não repetidos das duas listas.
- a primeira lista sem os elementos repetidos na segunda

**Exercício 6.20** Escreva um programa que compare duas listas. Considere a primeira lista como a versão inicial e a segunda como a versão após alterações. Utilizando operações com conjuntos, seu programa deverá imprimir a lista de modificações entre essas duas versões, listando:

- os elementos que não mudaram
- os novos elementos
- os elementos que foram removidos

## 6.22 Qual estrutura de dados utilizar?

Como programador, você vai passar boa parte do seu tempo tendo que decidir qual estrutura de dados utilizar em seus programas. Até agora, já vimos: listas, tuplas, dicionários e conjuntos. Um resumo:

*Tabela 6.3 – Resumo de estruturas de dados*

| | Listas | Tuplas | Dicionários | Conjuntos |
|---|---|---|---|---|
| Ordem dos elementos | Fixa | Fixa | Mantida a partir do Python 3.7 | Indeterminada |
| Tamanho | Variável | Fixo | Variável | Variável |
| Elementos repetidos | Sim | Sim | Pode repetir valores, mas as chaves devem ser únicas | Não |
| Pesquisa | Sequencial, índice numérico | Sequencial, índice numérico | Direta por chave | Direta por valor |
| Alterações | Sim | Não | Sim | Sim |
| Uso primário | Sequências | Sequências constantes | Dados indexados por chave | Verificação de unicidade, operações com conjuntos |

# CAPÍTULO 7

# Trabalhando com strings

No Capítulo 3, vimos que podemos acessar strings como listas, mas também falamos que strings são imutáveis em Python. Vejamos o que acontece se tentarmos alterar uma string:

```
>>> S = "Alô mundo"
>>> print(S[0])
A
>>> S[0] = "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Se quisermos trabalhar caractere a caractere com uma string, alterando seu valor, teremos de primeiro transformá-la em uma lista:

```
>>> L = list("Alô Mundo")
>>> L[0] = "a"
>>> print(L)
['a', 'l', 'ô', ' ', 'M', 'u', 'n', 'd', 'o']
>>> s = "".join(L)
>>> print(s)
alô Mundo
```

A função **list** transforma cada caractere da string em um elemento da lista retornada. Já o método `join` faz a operação inversa, transformando os elementos da lista em string.

## 7.1 Verificação parcial de strings

Quando você precisar verificar se uma string começa ou termina com alguns caracteres, pode usar os métodos `startswith` e `endswith`. Esses métodos verificam

apenas os primeiros (`startswith`) ou os últimos (`endswith`) caracteres da string, retornando **`True`** caso sejam iguais ou **`False`**, caso contrário.

```
>>> nome = "João da Silva"
>>> nome.startswith("João")
True
>>> nome.startswith("joão")
False
>>> nome.endswith("Silva")
True
```

Veja que comparamos "João da Silva" com "joão" e obtivemos **`False`**. Esse é um detalhe pequeno, mas importante, pois `startswith` e `endswith` consideram letras maiúsculas e minúsculas como letras diferentes.

Você pode resolver esse tipo de problema convertendo a string para maiúsculas ou minúsculas antes de realizar a comparação. O método `lower` retorna uma cópia da string com todos os caracteres minúsculos, e o método `upper` retorna uma cópia com todos os caracteres maiúsculos.

```
>>> s = "O Rato roeu a roupa do Rei de Roma"
>>> s.lower()
'o rato roeu a roupa do rei de roma'
>>> s.upper()
'O RATO ROEU A ROUPA DO REI DE ROMA'
>>> s.lower().startswith("o rato")
True
>>> s.upper().startswith("O RATO")
True
```

Não se esqueça de comparar com uma string onde todos os caracteres são maiúsculos ou minúsculos, dependendo se você utilizou `upper` ou `lower`, respectivamente.

Outra forma de verificar se uma palavra pertence a uma string é utilizando o operador **`in`**:

```
>>> s = "Maria Amélia Souza"
>>> "Amélia" in s
True
>>> "Maria" in s
True
>>> "Souza" in s
True
>>> "a A" in s
```

```
True
>>> "amélia" in s
False
```

Você também pode testar se uma string não está contida em outra, utilizando **not in**:

```
>>> s = "Todos os caminhos levam a Roma"
>>> "levam" not in s
False
>>> "Caminhos" not in s
True
>>> "AS" not in s
True
```

Veja que aqui também letras maiúsculas e minúsculas são diferentes. Você pode combinar `lower` e `upper` com **in** e **not in** para ignorar esse tipo de diferença:

```
>>> s = "João comprou um carro"
>>> "joão" in s.lower()
True
>>> "CARRO" in s.upper()
True
>>> "comprou" not in s.lower()
False
>>> "barco" not in s.lower()
True
```

O operador **in** também pode ser utilizado com listas normais, facilitando, assim, a pesquisa de elementos dentro de uma lista.

## 7.2 Contagem

Se você precisar contar as ocorrências de uma letra ou palavra em uma string, utilize o método `count`:

```
>>> t = "um tigre, dois tigres, três tigres"
>>> t.count("tigre")
3
>>> t.count("tigres")
2
>>> t.count("t")
4
>>> t.count("z")
0
```

## 7.3 Pesquisa de strings

Para pesquisar se uma string está dentro de outra e obter a posição da primeira ocorrência, você pode utilizar o método `find`:

```
>>> s = "Alô mundo"
>>> s.find("mun")
4
>>> s.find("ok")
-1
```

Caso a string seja encontrada, você obterá um valor maior ou igual a zero, ou `-1`, caso contrário. Observe que o valor retornado, quando maior ou igual a zero, é igual ao índice que pode ser utilizado para obter o primeiro caractere da string procurada.

Se o objetivo for pesquisar, mas da direita para a esquerda, utilize o método `rfind`, que realiza essa tarefa:

```
>>> s = "Um dia de sol"
>>> s.rfind("d")
7
>>> s.find("d")
3
```

Tanto `find` quanto `rfind` suportam duas opções adicionais: início (*start*) e fim (*end*). Se você especificar início, a pesquisa começará a partir dessa posição. Se especificar o fim, a pesquisa utilizará essa posição como último caractere a considerar na pesquisa. Por exemplo:

```
>>> s = "um tigre, dois tigres, três tigres"
>>> s.find("tigres")
15
>>> s.rfind("tigres")
28
>>> s.find("tigres", 7)  # início=7
15
>>> s.find("tigres", 30)  # início=30
-1
>>> s.find("tigres", 0, 10)  # início=0 fim=10
-1
```

Podemos usar o valor retornado por `find` e `rfind` para achar todas as ocorrências da string. Por exemplo:

```
# Programa 7.1 - Pesquisa de todas as ocorrências
s = "um tigre, dois tigres, três tigres"
p = 0
while(p > -1):
    p = s.find("tigre", p)
    if p >= 0:
        print(f"Posição: {p}")
        p += 1
```

Produz como resultado:

```
Posição: 3
Posição: 15
Posição: 28
```

Os métodos `index` e `rindex` são bem parecidos com `find` e `rfind`, respectivamente. A maior diferença é que, se a substring não for encontrada, `index` e `rindex` lançam uma exceção do tipo `ValueError`.

**Exercício 7.1** Escreva um programa que leia duas strings. Verifique se a segunda ocorre dentro da primeira e imprima a posição de início.

1ª string: AABBEFAATT

2ª string: BE

Resultado: BE encontrado na posição 3 de AABBEFAATT

**Exercício 7.2** Escreva um programa que leia duas strings e gere uma terceira com os caracteres comuns às duas strings lidas.

1ª string: AAACTBF

2ª string: CBT

Resultado: CBT

A ordem dos caracteres da string gerada não é importante, mas deve conter todas as letras comuns a ambas.

**Exercício 7.3** Escreva um programa que leia duas strings e gere uma terceira apenas com os caracteres que aparecem em uma delas.

1ª string: CTA

2ª string: ABC

3ª string: BT

A ordem dos caracteres da terceira string não é importante.

**Exercício 7.4** Escreva um programa que leia uma string e imprima quantas vezes cada caractere aparece nessa string.

String: TTAAC

Resultado:

T: 2x

A: 2x

C: 1x

**Exercício 7.5** Escreva um programa que leia duas strings e gere uma terceira, na qual os caracteres da segunda foram retirados da primeira.

1ª string: AATTGGAA

2ª string: TG

3ª string: AAAA

**Exercício 7.6** Escreva um programa que leia três strings. Imprima o resultado da substituição na primeira, dos caracteres da segunda pelos da terceira.

1ª string: AATTCGAA

2ª string: TG

3ª string: AC

Resultado: AAAACCAA

## 7.4 Posicionamento de strings

Python também traz métodos que ajudam a apresentar strings de formas mais interessantes. Vejamos o método `center`, que centraliza a string em um número de posições passado como parâmetro, preenchendo com espaços à direita e à esquerda até que a string esteja centralizada.

```
>>> s = "tigre"
>>> "X" + s.center(10) + "X"
X  tigre   X
>>> "X" + s.center(10, ".") + "X"
X..tigre...X
```

Se, além do tamanho, você também passar o caractere de preenchimento, este será utilizado no lugar de espaços em branco.

Se o que você deseja é apenas completar a string com espaços à esquerda, pode utilizar o método `ljust`. Se deseja completar com espaços à direita, utilize `rjust`:

```
>>> s = "tigre"
>>> s.ljust(20)
'tigre               '
>>> s.rjust(20)
'               tigre'
>>> s.ljust(20, ".")
'tigre..............'
>>> s.rjust(20, "-")
'---------------tigre'
```

Essas funções são úteis quando precisamos criar relatórios ou simplesmente alinhar a saída dos programas.

## 7.5 Quebra ou separação de strings

O método `split` quebra uma string a partir de um caractere passado como parâmetro, retornando uma lista com as substrings já separadas:

```
>>> s = "um tigre, dois tigres, três tigres"
>>> s.split(",")
['um tigre', ' dois tigres', ' três tigres']
>>> s.split(" ")
['um', 'tigre,', 'dois', 'tigres,', 'três', 'tigres']
>>> s.split()
['um', 'tigre,', 'dois', 'tigres,', 'três', 'tigres']
```

Observe que o caractere que utilizamos para dividir a string não é retornado na lista, ou seja, ele é utilizado para separar a string e depois descartado.

Se você deseja separar uma string, com várias linhas de texto, pode utilizar o método `splitlines`:

```
>>> m = "Uma linha\noutra linha\ne mais uma\n"
>>> m.splitlines()
['Uma linha', 'outra linha', 'e mais uma']
```

## 7.6 Substituição de strings

Para substituir trechos de uma string por outros, utilize o método `replace`. Com o método `replace`, o primeiro parâmetro é a string a substituir; e o segundo, o conteúdo que a substituirá. Opcionalmente, podemos passar um terceiro parâmetro que limita quantas vezes queremos realizar a repetição:

```
>>> s = "um tigre, dois tigres, três tigres"
>>> s.replace("tigre", "gato")
'um gato, dois gatos, três gatos'
>>> s.replace("tigre", "gato", 1)
'um gato, dois tigres, três tigres'
>>> s.replace("tigre", "gato", 2)
'um gato, dois gatos, três tigres'
>>> s.replace("tigre", "")
'um , dois s, três s'
>>> s.replace("", "-")
'-u-m- -t-i-g-r-e-,- -d-o-i-s- -t-i-g-r-e-s-,- -t-r-ê-s- -t-i-g-r-e-s-'
```

Se você passar uma string vazia no segundo parâmetro, o trecho será apagado. Se o primeiro parâmetro for uma string vazia, o segundo será inserido antes de cada caractere da string.

## 7.7 Remoção de espaços em branco

O método `strip` é utilizado para remover espaços em branco do início ou fim da string. Já os métodos `lstrip` e `rstrip` removem apenas os caracteres em branco à esquerda ou à direita, respectivamente:

```
>>> t = "    Olá   "
>>> t.strip()
'Olá'
>>> t.lstrip()
'Olá   '
>>> t.rstrip()
'    Olá'
```

Se você passar um parâmetro tanto para `strip` quanto para `lstrip` ou `rstrip`, este será utilizado como caractere a remover:

```
>>> s = "...///Olá///..."
>>> s.lstrip(".")
'///Olá///...'
```

```
>>> s.rstrip(".")
'...///Olá///'
>>> s.strip(".")
'///Olá///'
>>> s.strip("./")
'Olá'
```

## 7.8 Validação por tipo de conteúdo

Strings em Python podem ter seu conteúdo analisado e verificado utilizando-se métodos especiais. Esses métodos verificam se todos os caracteres são letras, números ou uma combinação deles. Vejamos alguns exemplos:

```
>>> s = "125"
>>> p = "alô mundo"
>>> s.isalnum()
True
>>> p.isalnum()
False
>>> s.isalpha()
False
>>> p.isalpha()
False
```

O método `isalnum` retorna verdadeiro se a string não estiver vazia, e se todos os seus caracteres são letras e/ou números. Se a string contiver outros tipos de caracteres, como espaços, vírgula, exclamação, interrogação ou caracteres de controle, retorna **`False`**.

Já `isalpha` é mais restritivo, retornando verdadeiro apenas se todos os caracteres forem letras, incluindo vogais acentuadas. Retorna falso se algum outro tipo de caractere for encontrado na string ou se estiver vazia.

O método `isdigit` verifica se o valor consiste em números, retornando **`True`** se a string não estiver vazia e contiver apenas números. Se a string contiver espaços, pontos, vírgulas ou sinais (+ ou -), retorna falso:

```
>>> "771".isdigit()
True
>>> "10.4".isdigit()
False
>>> "+10".isdigit()
```

```
False
>>> "-5".isdigit()
False
```

Os métodos `isdigit` e `isnumeric` são parecidos, e diferenciá-los envolve um conhecimento de Unicode mais aprofundado. `isdigit` retorna **`True`** para caracteres definidos como dígitos numéricos em Unicode, indo além de nossos 0 a 9, como o número 9 tibetano (`\u0f29`). `isnumeric` é mais abrangente, incluindo dígitos e representações numéricas como frações, por exemplo, ⅓ (`\u2153`). Vejamos alguns exemplos:

```
>>> umterço = "\u2153"
>>> novetibetano = "\u0f29"
>>> umterço.isdigit()
False
>>> umterço.isnumeric()
True
>>> novetibetano.isdigit()
True
>>> novetibetano.isnumeric()
True
```

Podemos também verificar se todos os caracteres de uma string são letras maiúsculas ou minúsculas usando `isupper` e `islower`, respectivamente:

```
>>> s = "ABC"
>>> p = "abc"
>>> e = "aBc"
>>> s.isupper()
True
>>> s.islower()
False
>>> p.isupper()
False
>>> p.islower()
True
>>> e.isupper()
False
>>> e.islower()
False
```

Temos também como verificar se a string contém apenas caracteres em branco, como espaços, marcas de tabulação (TAB), quebras de linha (LF) ou retorno de carro (CR). Para isso, vamos utilizar o método `isspace`:

```
>>> "\t\n\r    ".isspace()
True
>>> "\tAlô".isspace()
False
```

Se você precisar verificar se algo pode ser impresso na tela, o método `isprintable` pode ajudar. Ele retorna **`False`** se algum caractere que não pode ser impresso for encontrado na string. Você pode utilizá-lo para verificar se a impressão de uma string pode causar efeitos indesejados no terminal ou na formatação de um arquivo:

```
>>> "\n\t".isprintable()
False
>>> "\nAlô".isprintable()
False
>>> "Alô mundo".isprintable()
True
```

## 7.9 Formatação de strings

Já utilizamos f-strings e já vimos como o método **`format`** pode ser utilizado. Vejamos algumas opções extras para o método **`format`**.

Usando **`format`**, podemos substituir os valores usando números entre chaves. Cada número representa a posição do valor a substituir. A posição é dada pela sequência em que os parâmetros são listados na chamada de **`format`**:

```
>>> "{0} {1}".format("Alô", "Mundo")
'Alô Mundo'
>>> "{0} x {1} R${2}".format(5, "maçã", "1.20")
'5 x maçã R$1.20'
```

O número entre chaves é uma referência aos parâmetros passados ao método **`format`**, em que 0 é o primeiro parâmetro; 1, o segundo; e assim por diante, como os índices de uma lista. Uma das vantagens da nova sintaxe é utilizar o mesmo parâmetro várias vezes na mesma string:

```
>>> "{0} {1} {0}".format("-", "x")
'- x -'
```

Isso também permite a completa reordenação da mensagem, como imprimir os parâmetros em outra ordem:

```
>>> "{1} {0}".format("primeiro", "segundo")
'segundo primeiro'
```

Essa sintaxe permite também especificar a largura de cada valor, utilizando o símbolo de dois pontos (:) após a posição do parâmetro, como `0:10` no exemplo a seguir:

```
>>> "{0:10} {1}".format("123", "456")
'123        456'
>>> "X{0:10}X".format("123")
'X123       X'
>>> "X{0:10}X".format("123456789012345")
'X123456789012345X'
```

Que significa: substitua o primeiro parâmetro com uma largura de 10 caracteres. Se o primeiro parâmetro for menor que o tamanho informado, espaços serão utilizados para completar as posições que faltam. Se o parâmetro for maior que o tamanho especificado, ele será impresso em sua totalidade, ocupando mais espaço que o inicialmente especificado.

Podemos também especificar se queremos os espaços adicionais à esquerda ou à direita do valor, utilizando os símbolos de maior (>) e menor (<) logo depois dos dois pontos:

```
>>> "X{0:<10}X".format("123")
'X123       X'
>>> "X{0:>10}X".format("123")
'X       123X'
```

Se quisermos o valor entre os espaços, de forma a centralizá-lo, podemos utilizar o circunflexo (^):

```
>>> "X{0:^10}X".format("123")
'X   123    X'
```

Se quisermos outro caractere no lugar de espaços, podemos especificá-lo logo após os dois pontos:

```
>>> "X{0:.<10}X".format("123")
'X123.......X'
>>> "X{0:!>10}X".format("123")
'X!!!!!!!123X'
>>> "X{0:*^10}X".format("123")
'X***123****X'
```

Se o parâmetro for uma lista, podemos especificar o índice do elemento a substituir, dentro da máscara:

```
>>> "{0[0]} {0[1]}".format(["123", "456"])
'123 456'
```

O mesmo é válido para dicionários:

```
>>> "{0[nome]} {0[telefone]}".format({"telefone": 572, "nome": "Maria"})
'Maria 572'
```

Observe que dentro da string escrevemos nome e telefone entre colchetes, mas sem aspas, como normalmente faríamos ao utilizar dicionários. Essa sintaxe é especial para o método **format**.

A partir da versão 3.6 do Python, você pode utilizar a mesma formatação do método **format** com f-strings. Para converter uma chamada ao método **format** para uma f-string, escreva o conteúdo dentro de {} e substitua os números (exemplo: {0} pelas variáveis ou expressões passadas como parâmetros). Exemplo:

```
>>> "X{0:.<10}X".format("123")
'X123.......X'
>>> f"X{123:.<10}X"
'X123.......X'
```

Quando utilizar **format** ou f-strings dependerá de cada programa. Para casos mais simples, a sintaxe com f-strings é bem clara. Em casos mais complexos, como quando há várias repetições do mesmo parâmetro, o método **format** pode ser mais interessante. O método **format** também é mais compatível com versões mais antigas do Python.

### 7.9.1 Formatação de números

A nova sintaxe também permite a formatação de números. Por exemplo, se especificarmos o tamanho a imprimir com um zero à esquerda, o valor será impresso com a largura determinada e com zeros à esquerda completando o tamanho:

```
>>> "{0:05}".format(5)
'00005'
```

Podemos também utilizar outro caractere, diferente de 0, mas, nesse caso, devemos escrever o caractere à esquerda do símbolo de igualdade:

```
>>> "{0:*=7}".format(32)
'*****32'
```

Podemos também especificar o alinhamento dos números que estamos imprimindo usando <, > e ^:

```
>>> "{0:*^10}".format(123)
'***123****'
```

```
>>> "{0:*<10}".format(123)
'123*******'
>>> "{0:*>10}".format(123)
'*******123'
```

Podemos também utilizar uma vírgula para solicitar o agrupamento por milhar, e o ponto para indicar a precisão de números decimais, ou melhor, a quantidade de casas após a vírgula:

```
>>> "{0:10,}".format(7532)
'     7,532'
>>> "{0:10.5f}".format(1500.31)
'1500.31000'
>>> "{0:10,.5f}".format(1500.31)
'1,500.31000'
```

Esta sintaxe também permite forçar a impressão de sinais ou apenas reservar espaço para uma impressão eventual:

```
>>> "{0:+10} {1:-10}".format(5, -6)
'        +5         -6'
>>> "{0:-10} {1: 10}".format(5, -6)
'         5         -6'
>>> "{0: 10} {1:+10}".format(5, -6)
'         5         -6'
```

Quando trabalhamos com formatos numéricos, devemos indicar com uma letra o formato que deve ser adotado para a impressão. Essa letra informa como devemos exibir um número. A lista completa de formatos numéricos é apresentada nas tabelas 7.1 e 7.2.

*Tabela 7.1 – Formatos de números inteiros*

| Código | Descrição | Exemplo (45) |
|---|---|---|
| b | Binário | 101101 |
| c | Caractere | - |
| d | Base 10 | 45 |
| n | Base 10 local | 45 |
| o | Octal | 55 |
| x | Hexadecimal com letras minúsculas | 2d |
| X | Hexadecimal com letras maiúsculas | 2D |

*Tabela 7.2 – Formatos de números decimais*

| Código | Descrição | Exemplo (1.345) |
|---|---|---|
| e | Notação científica com e minúsculo | 1.345000e+00 |
| E | Notação científica com e maiúsculo | 1.345000E+00 |
| f | Decimal | 1.345000 |
| g | Genérico | 1.345 |
| G | Genérico | 1.345 |
| n | Local | 1,345 |
| % | Percentual | 134.500000% |

O formato `b` imprime o número utilizando o sistema binário, ou seja, de base 2, com apenas 0 e 1 como dígitos. O formato `o` imprime o número utilizando o sistema octal, ou seja, de base 8, com dígitos de 0 a 7. Já o formato `c` imprime o número convertendo-o em caractere, utilizando a tabela Unicode. Tanto o formato `x` quando o `X` imprimem os números utilizando o sistema hexadecimal, base 16. A diferença é que `x` utiliza letras minúsculas, e `X`, maiúsculas. Vejamos exemplos:

```
>>> "{:b}".format(5678)
'1011000101110'
>>> "{:c}".format(65)
'A'
>>> "{:o}".format(5678)
'13056'
>>> "{:x}".format(5678)
'162e'
>>> "{:X}".format(5678)
'162E'
```

Os formatos `d` e `n` são parecidos. O formato `d` é semelhante ao que utilizamos ao formatar os números com `%d`. A diferença entre o formato `d` e o `n` é que o `n` leva em consideração as configurações regionais da máquina do usuário.

Vejamos outro exemplo a seguir. Observe que, antes de configurarmos a máquina para o português do Brasil, o resultado de `d` e `n` eram iguais. Após a configuração regional, o formato `n` passou a exibir os números utilizando pontos para separar os milhares.

```
>>> "{:d}".format(5678)
'5678'
>>> "{:n}".format(5678)
'5678'
```

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "pt_BR.utf-8")
'pt_BR.utf-8'
>>> "{:n}".format(5678)
'5.678'
```

Para números decimais, temos também vários códigos. O código `f` já conhecemos e funciona de forma semelhante ao que utilizamos em `%f`. O formato `n` utiliza as configurações regionais para imprimir o número. Em português, essa configuração utiliza o ponto como separador de milhar e a vírgula como separador decimal, produzindo números mais fáceis de entender:

```
>>> "{:f}".format(1579.543)
'1579.543000'
>>> "{:n}".format(1579.543)
'1579.54'
>>> import locale
>>> locale.setlocale(locale.LC_ALL, "pt_BR.utf-8")
'pt_BR.utf-8'
>>> "{:n}".format(1579.543)
'1.579,54'
```

Os formatos `e` e `E` imprimem o número utilizando notação científica. Isso quer dizer que a parte decimal vai ser substituída por um expoente. Por exemplo: 1004.5 em notação científica será impresso 1.004500e+03. Para entender esse formato, devemos entender suas partes. O número à esquerda de `e` é o que chamamos de mantissa, e o número à direita, o expoente. A base é sempre 10 e, para recompor o número, multiplicamos a mantissa pela base 10 elevada ao expoente. No caso de 1.004500e+03, teríamos:

$$1.004500 \times 10^3 = 1.004500 \times 1000 = 1004.5$$

A vantagem de utilizar notação científica é representar números muito grandes, ou muito pequenos, em pouco espaço. A diferença entre o formato `e` e o `E` é que queremos exibir o `e` que separa a mantissa do expoente, ou seja, um `e` minúsculo ou um `E` maiúsculo. Os formatos `g` e `G` são chamados de genéricos, pois, dependendo do número, são exibidos como no formato `f` ou como no `e` ou `E`. O tipo `%` simplesmente multiplica o valor por 100 antes de imprimi-lo, assim, 0,05 é impresso como 5%. Veja alguns exemplos:

```
>>> "{:8e}".format(3.141592653589793)
'3.141593e+00'
>>> "{:8E}".format(3.141592653589793)
'3.141593E+00'
```

```
>>> "{:8g}".format(3.141592653589793)
' 3.14159'
>>> "{:8G}".format(3.141592653589793)
' 3.14159'
>>> "{:8g}".format(3.14)
'    3.14'
>>> "{:8G}".format(3.14)
'    3.14'
>>> "{:5.2%}".format(0.05)
'5.00%'
```

## 7.10 Jogo da forca

Vejamos um jogo muito simples, mas que ajuda a trabalhar com strings. O jogo da forca é simples e pode até se tornar divertido.

```
# Programa 7.2 - Jogo da forca
palavra = input("Digite a palavra secreta:").lower().strip() ❶
for x in range(100):
    print() ❷
digitadas = []
acertos = []
erros = 0
while True:
    senha = ""
    for letra in palavra:
        senha += letra if letra in acertos else "." ❸
    print(senha)
    if senha == palavra:
        print("Você acertou!")
        break
    tentativa = input("\nDigite uma letra:").lower().strip()
    if tentativa in digitadas:
        print("Você já tentou esta letra!")
        continue ❹
    else:
        digitadas += tentativa
        if tentativa in palavra:
            acertos += tentativa
        else:
```

```
            erros += 1
            print("Você errou!")
    print("X==:==\nX  :  ")
    print("X  O  " if erros >= 1 else "X")
    linha2 = ""
    if erros == 2:
        linha2 = "  |  "
    elif erros == 3:
        linha2 = " \|  "
    elif erros >= 4:
        linha2 = " \|/ "
    print(f"X{linha2}")
    linha3 = ""
    if erros == 5:
        linha3 += " /   "
    elif erros >= 6:
        linha3 += " / \ "
    print(f"X{linha3}")
    print("X\n==========")
    if erros == 6:
        print("Enforcado!")
        break
```

Veja como é simples escrever um jogo da forca em Python. Vamos analisar o que essa listagem faz. Em ❶, aproveitamos o retorno de **input** para chamar os métodos de string `lower` e `strip`. Observe como escrevemos uma chamada após a outra. Isso é possível porque **input**, `lower` e `strip` retornam um objeto `string`. O resultado final da string digitada pelo usuário, convertida para letras minúsculas e com espaços em branco no início e no fim eliminados, é atribuído à variável `palavra`. É essa variável que vai armazenar a palavra a ser adivinhada pelo jogador.

Em ❷ pulamos várias linhas para que o jogador não veja o que foi digitado como `palavra`. A ideia é que um jogador escreva uma palavra secreta, e que outro tente descobri-la.

Em ❸, temos uma nova forma de condição que facilita decisões simples. O **if** imediato, ou na mesma linha, serve para decidir o valor a retornar, dependendo de uma condição. É como o **if** que já conhecemos, mas o valor verdadeiro fica à esquerda do **if**, e a condição, à sua direita. O **else** não tem : e indica o resultado caso a condição seja falsa. No exemplo, letra é o valor a retornar se a condição `letra` **in** `acertos` for verdadeira, e "." é o valor se o resultado for falso. Assim:

```
senha += letra if letra in acertos else "."
```

substitui:

```
if letra in acertos:
    senha += letra
else:
    senha += "."
```

A vantagem dessa construção é que escrevemos tudo em uma só linha. Esse recurso é interessante para expressões simples e não deve ser utilizado para todo e qualquer caso. No jogo da forca, usamos essa construção para mostrar a palavra secreta na tela, mas substituindo todas as letras ainda não adivinhadas por um ".".

Em ❹, utilizamos a instrução **continue**. A instrução **continue** é similar ao **break** que já conhecemos, mas serve para indicar que devemos ignorar todas as linhas até o fim da repetição e voltar para o início, sem terminá-la. No jogo da forca, isso faz com que a execução passe de ❹ para o **while** diretamente, pulando todas as linhas depois dela. Quando utilizada com **while**, **continue** causa a reavaliação da condição da repetição; quando utilizada com **for**, faz com que o próximo elemento seja utilizado. Considere **continue** como vá para o fim da repetição e volte para a linha de **for** ou **while**.

**Exercício 7.7** Modifique o o jogo da forca (Programa 7.2) de forma a escrever a palavra secreta caso o jogador perca.

**Exercício 7.8** Modifique o Programa 7.2 de forma a utilizar uma lista de palavras. No início, pergunte um número e calcule o índice da palavra a utilizar pela fórmula: `indice = (número * 776) % len(lista_de_palavras)`.

**Exercício 7.9** Modifique o Programa 7.2 para utilizar listas de strings para desenhar o boneco da forca. Você pode utilizar uma lista para cada linha e organizá-las em uma lista de listas. Em vez de controlar quando imprimir cada parte, desenhe nessas listas, substituindo o elemento a desenhar.

Exemplo:

```
>>> linha = list("X------")
>>> linha
['X', '-', '-', '-', '-', '-', '-']
>>> linha[6] = "|"
>>> linha
['X', '-', '-', '-', '-', '-', '|']
>>> "".join(linha)
'X-----|'
```

**Exercício 7.10** Escreva um jogo da velha para dois jogadores. O jogo deve perguntar onde você quer jogar e alternar entre os jogadores. A cada jogada, verifique se a posição está livre. Verifique também quando um jogador venceu a partida. Um jogo da velha pode ser visto como uma lista de 3 elementos, na qual cada elemento é outra lista, também com três elementos.

Exemplo do jogo:

```
 X | O |
---+---+---
   | X | X
---+---+---
   |   | O
```

Em que cada posição pode ser vista como um número. Confira a seguir um exemplo das posições mapeadas para a mesma posição de seu teclado numérico.

```
 7 | 8 | 9
---+---+---
 4 | 5 | 6
---+---+---
 1 | 2 | 3
```

# CAPÍTULO 8

# Funções

Podemos definir nossas próprias funções em Python. Já sabemos como usar várias funções, como **len**, **int**, **float**, **print** e **input**. Neste capítulo, veremos como declarar novas funções e utilizá-las em programas.

Para definir uma nova função, utilizaremos a instrução **def**. Vejamos como declarar uma função de soma que recebe dois números como parâmetros e os imprime na tela:

```
def soma(a, b): ❶
    print(a + b) ❷
soma(2, 9) ❸
soma(7, 8)
soma(10, 15)
```

Observe em ❶ que usamos a instrução **def** seguida pelo nome da função, no caso, `soma`. Após o nome e entre parênteses, especificamos o nome dos parâmetros que a função receberá. Chamamos o primeiro de `a` e o segundo de `b`. Observe também que usamos `:` após os parâmetros para indicar o início de um bloco.

Em ❷, usamos a função **print** para exibir `a + b`. Observe que escrevemos ❷ dentro do bloco da função, ou seja, mais à direita.

Diferentemente do que já vimos até agora, essas linhas não serão executadas imediatamente, exceto a definição da função em si. Na realidade, a definição prepara o interpretador para executar a função quando esta for chamada em outras partes do programa. Para chamar uma função definida no programa, faremos da mesma forma que as funções já definidas na linguagem, ou seja, nome da função seguido dos parâmetros entre parênteses. Exemplos de como chamar a função `soma` são apresentados a partir de ❸. No primeiro exemplo, chamamos `soma(2, 9)`. Nesse caso, a função será chamada com `a` valendo 2, e `b` valendo 9. Os parâmetros são substituídos na mesma ordem em que foram definidos, ou seja, o primeiro valor como `a` e o segundo como `b`.

Funções são especialmente interessantes para isolar uma tarefa específica em um trecho de programa. Isso permite que a solução de um problema seja reutilizada em outras partes do programa, sem precisar repetir as mesmas linhas. O exemplo anterior utiliza dois parâmetros e imprime sua soma. Essa função não retorna valores como a função **len** ou a **int**. Vamos reescrever essa função de forma que o valor da soma seja retornado:

```
def soma(a, b):
    return a + b ❶
print(soma(2, 9))
```

Veja que agora utilizamos a instrução **return** ❶ para indicar o valor a retornar. Observe também que retiramos o **print** da função. Isso é interessante porque a soma e a impressão da soma de dois números são dois problemas diferentes. Nem sempre vamos somar e imprimir a soma, por isso, vamos deixar a função realizar apenas o cálculo. Se precisarmos imprimir o resultado, podemos utilizar a função **print**, como no exemplo.

Vejamos outro exemplo, uma função que retorne verdadeiro ou falso, dependendo se o número é par ou ímpar:

```
def épar(x):
    return x % 2 == 0
print(épar(2))
print(épar(3))
print(épar(10))
```

Imagine agora que precisamos definir uma função para retornar a palavra par ou ímpar. Podemos reutilizar `épar` em outra função:

```
def épar(x):
    return x % 2 == 0
def par_ou_ímpar(x):
    if épar(x): ❶
        return "par" ❷
    else:
        return "ímpar" ❸
print(par_ou_ímpar(4))
print(par_ou_ímpar(5))
```

Em ❶ chamamos a função `épar` dentro da função `par_ou_ímpar`. Observe que não há nada de especial nessa chamada: apenas repassamos o valor de `x` para a função `épar` e utilizamos seu retorno no **if**. Se a função retornar verdadeiro, retornaremos "par" ❷; caso contrário, "ímpar" em ❸. Utilizamos dois **return** na

função `par_ou_ímpar`. A instrução **`return`** faz com que a função pare de executar e que o valor seja retornado imediatamente ao programa ou à função que a chamou. Assim, podemos entender a instrução **`return`** como uma interrupção da execução da função, seguida do retorno do valor. As linhas da função após a instrução **`return`** são ignoradas de forma similar à instrução **`break`** dentro de um **`while`** ou **`for`**.

**Exercício 8.1** Escreva uma função que retorne o maior de dois números.

Valores esperados:

```
máximo(5, 6) == 6
máximo(2, 1) == 2
máximo(7, 7) == 7
```

**Exercício 8.2** Escreva uma função que receba dois números e retorne **`True`** se o primeiro número for múltiplo do segundo.

Valores esperados:

```
múltiplo(8, 4) == True
múltiplo(7, 3) == False
múltiplo(5, 5) == True
```

**Exercício 8.3** Escreva uma função que receba o lado de um quadrado e retorne sua área ($A = lado^2$).

Valores esperados:

```
área_quadrado(4) == 16
área_quadrado(9) == 81
```

**Exercício 8.4** Escreva uma função que receba a base e a altura de um triângulo e retorne sua área (A = (base x altura) / 2).

Valores esperados:

```
área_triângulo(6, 9) == 27
área_triângulo(5, 8) == 20
```

Vejamos agora um exemplo de função de pesquisa em uma lista:

```
# Programa 8.1 - Pesquisa em uma lista
def pesquise(lista, valor):
    for x, e in enumerate(lista):
        if e == valor:
```

```
            return x ❶
    return None ❷
L = [10, 20, 25, 30]
print(pesquise(L, 25))
print(pesquise(L, 27))
```

A função `pesquise` recebe dois parâmetros: a lista e o valor a pesquisar. Se o valor for encontrado, retornaremos o valor de sua posição em ❶. Caso não seja encontrado, retornaremos `None` em ❷. Observe que, se retornarmos em ❶, tanto `for` quanto o `return` em ❷ são completamente ignorados. Dizemos que `return` marca o fim da execução da função. Usando estruturas de repetição e condicionais, podemos programar onde e como as funções retornarão, assim como decidir o valor a ser retornado.

Vejamos outro exemplo, calculando a média dos valores de uma lista:

```
def soma(L):
    total = 0
    for e in L:
        total += e
    return total
def média(L):
    return soma(L) / len(L)
```

Para calcular a `média`, definimos outra função chamada `soma`. Dessa forma, teremos duas funções, uma para calcular a média e outra para calcular a soma da lista. Definir as funções dessa forma é mais interessante, pois uma função deve resolver apenas um problema. Poderíamos também ter definido a função `média` como:

```
def média(L):
    total = 0
    for e in L:
        total += e
    return total / len(L)
```

Qual das duas formas escolher depende do tipo de problema que se quer resolver. Se você não precisa do valor da soma dos elementos de uma lista em nenhuma parte do programa, a segunda forma é interessante. A primeira (soma e média definidas em duas funções separadas) é mais interessante em longo prazo e é também uma boa prática de programação. Conforme formos criando funções, podemos armazená-las para uso em outros programas.

Com o tempo, você vai colecionar essas funções, criando uma biblioteca ou módulo. Veremos mais sobre isso quando falarmos de importação e módulos.

Por enquanto, tente fixar duas regras: uma função deve resolver apenas um problema e, quanto mais genérica for sua solução, melhor ela será em longo prazo.

Para saber se sua função resolve apenas um problema, tente defini-la sem utilizar a conjunção "e". Se ela faz isso e aquilo, já é um sinal que efetua mais de uma tarefa e que talvez tenha de ser desmembrada em outras funções. Não se preocupe agora em definir funções perfeitas, pois, à medida que você for ganhando experiência na programação, esses conceitos se tornarão mais claros.

Resolver os problemas da maneira mais genérica possível é se preparar para reutilizar a função em outros programas. O fato de a função só ser utilizada no mesmo programa que a definiu não é um grande problema, pois, à medida que os programas se tornam mais complexos, eles exigem soluções próprias e detalhadas. No entanto, se todas as suas funções servirem apenas a um programa, considere isso como um sinal de alerta.

Vejamos o que não fazer:

```
# Programa 8.2 - Como não escrever uma função
def soma(L):
    total = 0
    x = 0
    while x < 5:
        total += L[x]
        x += 1
    return total
L = [1, 7, 2, 9, 15]
print(soma(L)) ❶
print(soma([7, 9, 12, 3, 100, 20, 4])) ❷
```

Você saberia dizer por que a função funcionou em ❶, mas retornou um valor incorreto em ❷?

A forma como definimos a função `soma` não é genérica, ou melhor, só funciona em casos em que devemos somar listas com cinco elementos. É para esse tipo de erro que você deve ficar atento. Se a função deve somar todos os elementos de qualquer lista passada como parâmetro, devemos ao menos presumir que podemos passar listas com tamanhos diferentes. Veremos mais sobre isso quando falarmos de testes.

Um problema clássico de programação é o cálculo do fatorial. O fatorial de um número é utilizado em estatística para calcular o número de combinações e permutações de conjuntos. Seu cálculo é simples, por isso, é muito utilizado como

exemplo em cursos de programação. Para calcular o fatorial, multiplicamos o número por todos os números que o precedem até chegarmos em 1.

Por exemplo:

- fatorial de 3: 3 x 2 x 1 = 6.
- fatorial de 4: 4 x 3 x 2 x 1 = 24

E assim por diante. Um caso especial é o fatorial de 0 que é definido como 1. Vejamos uma função que calcule o fatorial de um número:

```
# Programa 8.3 - Cálculo do fatorial
def fatorial(n):
    fat = 1
    while n > 1:
        fat *= n
        n -= 1
    return fat
```

Se você rastrear essa função, verá que calculamos o fatorial multiplicando o valor de `n` pelo valor anterior (`fat`) e que começamos do maior número até chegarmos em 1. Observe que a forma que definimos a função satisfaz o caso especial do fatorial de zero.

Poderíamos também ter definido a função como:

```
# Programa 8.4 - Outra forma de calcular o fatorial
def fatorial(n):
    fat = 1
    x = 1
    while x <= n:
        fat *= x
        x += 1
    return fat
```

Existem várias formas de resolver o problema, e todas podem estar corretas. Para decidir se nossa implementação está correta, temos de observar os valores retornados e compará-los aos valores de referência.

**Exercício 8.5** Reescreva a função do Programa 8.1 de forma a utilizar os métodos de pesquisa em lista, vistos no Capítulo 7.

**Exercício 8.6** Reescreva o Programa 8.2 de forma a utilizar **`for`** em vez de **`while`**.

DICA: Python tem funções para calcular a soma, o máximo e o mínimo de uma lista.

```
>>> L = [5, 7, 8]
>>> sum(L)
20
>>> max(L)
8
>>> min(L)
5
```

## 8.1 Variáveis locais e globais

Quando usamos funções, começamos a trabalhar com variáveis internas ou locais e com variáveis externas ou globais. A diferença entre elas é a visibilidade ou escopo.

Uma variável local a uma função existe apenas dentro dela, sendo normalmente inicializada a cada chamada. Assim, não podemos acessar o valor de uma variável local fora da função que a criou e, por isso, passamos parâmetros e retornamos valores nas funções, de forma a possibilitar a troca de dados no programa.

Uma variável global é definida fora de uma função, podendo ser vista por todas as funções do módulo (programa) e por todos os módulos que importam o módulo que a definiu.

Vejamos um exemplo dos dois casos:

```
EMPRESA = "Unidos Venceremos Ltda"
def imprime_cabeçalho():
    print(EMPRESA)
    print("-" * len(EMPRESA))
```

A função `imprime_cabeçalho` não recebe parâmetros nem retorna valores. Ela simplesmente imprime o nome da empresa e traços abaixo dele. Observe que utilizamos a variável `EMPRESA` definida fora da função. Nesse caso, `EMPRESA` é uma variável global, podendo ser acessada em qualquer função.

Variáveis globais devem ser utilizadas o mínimo possível em seus programas, pois dificultam a leitura e violam o encapsulamento da função. Aqui, a definição de encapsulamento é o fato de uma função conter ou esconder os detalhes de sua operação, de forma que seu funcionamento seja entendido analisando-se apenas

os parâmetros de entrada e o código da própria função. A dificuldade da leitura gerada pela utilização excessiva de variáveis globais é ter de procurar pela definição e conteúdos fora da função em si, que podem mudar entre diferentes chamadas. Além disso, uma variável global pode ser alterada por qualquer função, tornando a tarefa de saber quem altera seu valor realmente mais trabalhosa. O encapsulamento é comprometido porque a função depende de uma variável externa, ou seja, que não é declarada dentro da função nem recebida como parâmetro.

Embora devamos utilizar variáveis globais com cuidado, isso não significa que elas não tenham uso ou que possam simplesmente ser classificadas como má prática. Um bom uso de variáveis globais é guardar valores constantes e que devem ser acessíveis a todas as funções do programa, como o nome da empresa.

Elas também são utilizadas em nosso programa principal e para inicializar o módulo com valores iniciais. Tente utilizar variáveis globais apenas para configuração e com valores constantes. Por exemplo, se na função `imprime_cabeçalho` o nome da empresa mudasse entre uma chamada e outra, ele não deveria ser armazenado em uma variável global, mas passado por parâmetro.

Devido à capacidade da linguagem Python de declarar variáveis à medida que precisarmos, devemos tomar cuidado quando alteramos uma variável global dentro de uma função. Vejamos um exemplo:

```
a = 5 ❶
def muda_e_imprime():
    a = 7 ❷
    print(f"A dentro da função: {a}")
print(f"a antes de mudar: {a}") ❸
muda_e_imprime() ❹
print(f"a depois de mudar: {a}") ❺
```

Execute o programa e analise seu resultado. Você sabe por que o valor de `a` não mudou? Tente responder a essa questão antes de continuar lendo.

Em ❶, criamos a variável global `a`. Em ❷, temos a variável local da função, também chamada `a`, recebendo 7. Em ❸ e ❺, imprimimos o valor da variável global `a` e, em ❹, o valor da variável local `a`. Para o computador, essas variáveis são completamente diferentes, embora tenham o mesmo nome. Em ❹, podemos acessar seu conteúdo porque realizamos a impressão dentro da função. Da mesma forma, a variável `a` que imprimimos é a variável local, valendo 7. Essa variável local deixa de existir no final da função e explica por que não alteramos a variável global `a`.

Se quisermos modificar uma variável global dentro de uma função, devemos informar que estamos usando uma variável global antes de inicializá-la, na primeira linha de nossa função. Essa definição é feita com a instrução **global**. Vejamos o exemplo modificado:

```
a = 5
def muda_e_imprime():
    global a ❶
    a = 7 ❷
    print(f"A dentro da função: {a}")
print(f"a antes de mudar: {a}")
muda_e_imprime()
print(f"a depois de mudar: {a}")
```

Agora, em ❶, estamos trabalhando com a variável a global. Assim, quando fizemos a = 7 em ❷, trocamos o valor de a.

Lembre-se de limitar o uso de variáveis globais.

A utilização da instrução **global** pode sinalizar um problema de construção no programa. Lembre-se de utilizar variáveis globais apenas para configuração e, de preferência, como constantes. Dessa forma, você utilizará **global** apenas nas funções que alteram a configuração de seu programa ou módulo. A instrução **global** facilita a escolha de nomes de variáveis locais nos programas, pois deixa claro que estamos usando uma variável preexistente, e não criando uma nova.

Tente definir variáveis globais de forma a facilitar a leitura de seu programa, como usar apenas letras maiúsculas em seus nomes ou um prefixo, por exemplo, EMPRESA ou G_EMPRESA.

## 8.2 Funções recursivas

Uma função pode chamar a si mesma. Quando isso ocorre, temos uma função recursiva. O problema do fatorial é interessante para demonstrar o conceito de recursividade. Podemos definir o fatorial de um número como sendo esse número multiplicado pelo fatorial de seu antecessor. Se chamarmos nosso número de n, temos uma definição recursiva do fatorial como:

$$fatorial(n) = \begin{cases} 1 & 0 \leq n \leq 1 \\ n \times fatorial(n-1) & \end{cases}$$

Em Python, definiríamos a função recursiva para cálculo do fatorial como:

```
# Programa 8.5 - Função recursiva do fatorial
def fatorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * fatorial(n - 1)
```

Vamos adicionar algumas linhas para facilitar o rastreamento, alterando o programa para imprimir na entrada da função:

```
# Programa 8.6 - Função modificada para facilitar o rastreamento
def fatorial(n):
    print(f"Calculando o fatorial de {n}")
    if n == 0 or n == 1:
        print(f"Fatorial de {n} = 1")
        return 1
    else:
        fat = n * fatorial(n - 1)
        print(f" fatorial de {n} = {fat}")
    return fat
fatorial(4)
```

Que produz como resultado para o cálculo do fatorial de 4 a tela:

```
Calculando o fatorial de 4
Calculando o fatorial de 3
Calculando o fatorial de 2
Calculando o fatorial de 1
Fatorial de 1 = 1
 fatorial de 2 = 2
 fatorial de 3 = 6
 fatorial de 4 = 24
```

A sequência de Fibonacci é outro problema clássico no qual podemos aplicar funções recursivas. A sequência pode ser entendida como o número de casais de coelhos que teríamos após cada ciclo de reprodução, considerando que cada ciclo dá origem a um casal. Embora matematicamente a sequência de Fibonacci revele propriedades mais complexas, vamos nos limitar à história dos casais de coelhos.

A sequência começa com dois números 0 e 1. Os números seguintes são a soma dos dois anteriores. A sequência ficaria assim: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Assim, uma função para calcular o enésimo termo da sequência de Fibonacci pode ser definida como:

$$fibonacci(n) = \begin{cases} n & n \leq 1 \\ fibonacci(n-1) + fibonacci(n-2) \end{cases}$$

Sendo implementada em Python como:

```
# Programa 8.7 - Função recursiva de Fibonacci
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

**Exercício 8.7** Defina uma função recursiva que calcule o maior divisor comum (M.D.C.) entre dois números a e b, em que a > b.

$$mdc(a,b) = \begin{cases} a & b = 0 \\ mdc(b, a - b\left\lfloor\frac{a}{b}\right\rfloor) & a > b \end{cases}$$

Em que $a - b\left\lfloor\frac{a}{b}\right\rfloor$ pode ser escrito em Python como: `a % b`.

**Exercício 8.8** Usando a função mdc definida no exercício anterior, defina uma função para calcular o menor múltiplo comum (M.M.C.) entre dois números.

$$mmc(a,b) = \left\{ \frac{|a \times b|}{mdc(a,b)} \right.$$

Em que |a × b| pode ser escrito em Python como: **abs(a * b)**.

**Exercício 8.9** Rastreie o Programa 8.6 e compare o seu resultado com o apresentado.

**Exercício 8.10** Reescreva a função para cálculo da sequência de Fibonacci, sem utilizar recursão.

## 8.3 Validação

Funções são muito úteis para validar a entrada de dados. Vejamos o código para ler um valor inteiro, limitado por um valor de mínimo e máximo. Esse trecho repete a entrada de dados até termos um valor válido.

```
# Programa 8.8 - Exemplo de validação sem usar uma função
while True:
    v = int(input("Digite um valor entre 0 e 5:"))
    if v < 0 or v > 5:
        print("Valor inválido.")
    else:
        break
```

Podemos transformá-lo em uma função que receba a pergunta e os valores de máximo e mínimo:

```
# Programa 8.9 - Validação de inteiro usando função
def faixa_int(pergunta, mínimo, máximo):
    while True:
        v = int(input(pergunta))
        if v < mínimo or v > máximo:
            print(f"Valor inválido. Digite um valor entre {mínimo} e {máximo}")
        else:
            return v
```

Esse tipo de verificação é muito importante quando nosso programa só funciona com uma faixa de valores. Quando verificamos os dados do programa, estamos realizando uma validação. A validação é muito importante para evitarmos erros difíceis de detectar depois de termos escrito o programa. Sempre que seu programa receber dados, seja pelo teclado ou por um arquivo, verifique se esses dados estão na faixa e no formato adequado para o bom funcionamento.

**Exercício 8.11** Escreva uma função para validar uma variável string. Essa função recebe como parâmetro a string, o número mínimo e máximo de caracteres. Retorne verdadeiro se o tamanho da string estiver entre os valores de máximo e mínimo, e falso, caso contrário.

**Exercício 8.12** Escreva uma função que receba uma string e uma lista. A função deve comparar a string passada com os elementos da lista, também passada como parâmetro. Retorne verdadeiro se a string for encontrada dentro da lista, e falso, caso contrário.

**Exercício 8.13** Escreva uma função que receba uma string com as opções válidas a aceitar (cada opção é uma letra). Converta as opções válidas para letras minúsculas. Utilize **input** para ler uma opção, converter o valor para letras minúsculas e verificar se a opção é válida. Em caso de opção inválida, a função deve pedir ao usuário que digite novamente outra opção.

## 8.4 Parâmetros opcionais

Nem sempre precisaremos passar todos os parâmetros para uma função, preferindo utilizar um valor previamente escolhido como padrão, mas deixando a possibilidade de alterá-lo, caso necessário. Vejamos uma função que imprime uma barra na tela:

```
def barra():
    print("*" * 40)
```

Nesse exemplo, a função `barra` não recebe algum parâmetro e pode ser chamada com `barra()`. Porém, tanto o asterisco (*) quanto a quantidade de caracteres a exibir na barra podem precisar ser alterados. Por exemplo, podemos utilizar uma barra de asteriscos em uma parte de nosso programa e uma barra com pontos em outra. Para resolver esse problema, podemos utilizar parâmetros opcionais.

```
def barra(n=40, caractere="*"):
    print(caractere * n)
```

Essa nova definição permite utilizar a função `barra` como antes, ou seja, `barra()`, em que caractere será igual a `"*"` e `n` será igual a 40. Ao usarmos parâmetros opcionais, estamos especificando que valores devem ser utilizados caso um novo valor não seja especificado, mas ainda assim deixando a possibilidade de passarmos outro valor.

Passagem de parâmetros opcionais:

```
>>> barra(10)       # faz com que n seja 10
**********
>>> barra(10, "-")  # n = 10 e caractere = "-"
```

Parâmetros opcionais são úteis para evitar a passagem desnecessária dos mesmos valores, mas preservando a opção de passar valores, se necessário.

Podemos combinar parâmetros opcionais com obrigatórios na mesma função. Vejamos a função soma:

```
# Programa 8.10 - Função soma com parâmetros obrigatórios e opcionais
def soma(a, b, imprime=False):
    s = a + b
    if imprime:
        print(s)
    return s
```

No exemplo, `a` e `b` são parâmetros obrigatórios e `imprime` é um parâmetro opcional. Essa função pode ser utilizada como:

```
>>> soma(2, 3)
5
>>> soma(3, 4, True)
7
7
>>> soma(5, 8, False)
13
```

Embora possamos combinar parâmetros opcionais e obrigatórios, eles não podem ser misturados entre si, e os parâmetros opcionais devem sempre ser os últimos. Vejamos uma definição inválida:

```
def soma(imprime=True, a, b):
    s = a + b
    if imprime:
        print(s)
    return s
```

Essa definição é inválida porque o parâmetro opcional `imprime` é seguido por parâmetros obrigatórios `a` e `b`.

## 8.5 Nomeando parâmetros

Python suporta a chamada de funções com vários parâmetros, mas até agora vimos apenas o caso em que fizemos a chamada da função passando os parâmetros na mesma ordem em que foram definidos. Quando especificamos o nome dos parâmetros, podemos passá-los em qualquer ordem. Vejamos o exemplo da função `retângulo`, definida como:

```
# Programa 8.11 - Função retângulo com parâmetros obrigatórios e opcionais
def retângulo(largura, altura, caractere="*"):
    linha = caractere * largura
    for i in range(altura):
        print(linha)
```

A função `retângulo` pode ser chamada como:

```
>>> retângulo(3, 4)
***
***
***
***
>>> retângulo(largura=3, altura=4)
```

```
***
***
***
***
>>> retângulo(altura=4, largura=3)
***
***
***
***
>>> retângulo(caractere="-", altura=4, largura=3)
---
---
---
---
```

Uma vez que utilizamos o nome de cada parâmetro para fazer a chamada, a ordem de passagem deixa de ser importante. Observe também que o parâmetro `caractere` continua opcional, mas, se o especificarmos, devemos também o fazer com nome.

Quando especificamos o nome de um parâmetro, somos obrigados a especificar o nome de todos os outros parâmetros também. Por exemplo, as chamadas a seguir são inválidas:

```
retângulo(largura=3, 4)
retângulo(largura=3, altura=4, "*")
```

Uma exceção a essa regra é quando combinamos parâmetros obrigatórios e opcionais. Por exemplo, podemos passar os parâmetros obrigatórios sem usar seus nomes, mas respeitando a ordem usada na declaração e parâmetros nomeados apenas para escolher que parâmetros opcionais usar. Lembre-se de que, ao passar o primeiro parâmetro nomeado (usando seu nome), todos os parâmetros seguintes também devem ter seus nomes especificados.

## 8.6 Funções como parâmetro

Um poderoso recurso de Python é permitir a passagem de funções como parâmetro. Isso permite combinar várias funções para realizar uma tarefa. Vejamos um exemplo:

```
# Programa 8.12 - Funções como parâmetro
def soma(a, b):
    return a + b
```

```
def subtração(a, b):
    return a - b
def imprime(a, b, foper):
    print(foper(a, b)) ❶
imprime(5, 4, soma) ❷
imprime(10, 1, subtração) ❸
```

As funções `soma` e `subtração` foram definidas normalmente, mas a função `imprime` recebe um parâmetro chamado `foper`. Nesse caso, `foper` é a função que passaremos como parâmetro. Em ❶, passamos os parâmetros `a` e `b` para a função `foper` que ainda não conhecemos. Observe que passamos `a` e `b` a `foper` como faríamos com qualquer outra função. Em ❷, chamamos a função `imprime`, passando a função `soma` como parâmetro. Observe que, para passar a função `soma` como parâmetro, escrevemos apenas seu nome, sem passar qualquer parâmetro ou mesmo usar parênteses, como faríamos com uma variável normal. Em ❸, chamamos a função `imprime`, mas dessa vez passando a função `subtração` como `foper`.

Passar funções como parâmetro permite injetar funcionalidades dentro de outras funções, tornando-as configuráveis e mais genéricas. Vejamos outro exemplo:

```
# Programa 8.13 - Configuração de funções com funções
def imprime_lista(L, fimpressão, fcondição):
    for e in L:
        if fcondição(e):
            fimpressão(e)
def imprime_elemento(e):
    print(f"Valor: {e}")
def épar(x):
    return x % 2 == 0
def éimpar(x):
    return not épar(x)
L = [1, 7, 9, 2, 11, 0]
imprime_lista(L, imprime_elemento, épar)
imprime_lista(L, imprime_elemento, éimpar)
```

A função `imprime_lista` recebe uma lista e duas funções como parâmetro. A função `fimprime` é utilizada para realizar a impressão de cada elemento, mas só é chamada se o resultado da função passada como `fcondição` for verdadeiro. No exemplo, chamamos `imprime_lista` passando a função `épar` como parâmetro, fazendo com que apenas os elementos pares sejam impressos. Depois, passamos a função `éimpar`, de forma a imprimir apenas elementos cujo valor seja ímpar.

## 8.7 Empacotamento e desempacotamento de parâmetros

Outra flexibilidade da linguagem Python é passar parâmetros empacotados em uma lista. Vejamos um exemplo:

```
def soma(a, b):
    print(a + b)
L = [2, 3]
soma(*L) ❶
```

Em ❶, estamos utilizando o asterisco para indicar que queremos desempacotar a lista `L` utilizando seus valores como parâmetro para a função `soma`. No exemplo, `L[0]` será atribuído a `a` e `L[1]` a `b`. Esse recurso permite armazenar nossos parâmetros em listas e evita construções do tipo `soma(L[0], L[1])`.

Vejamos outro exemplo, em que utilizaremos uma lista de listas para realizar várias chamadas a uma função dentro de um **`for`**:

```
def barra(n=10, c="*"):
    print(c * n)
L = [[5, "-"], [10, "*"], [5], [6, "."]]
for e in L:
    barra(*e)
```

Observe que, mesmo usando o empacotamento de parâmetros, recursos como parâmetros opcionais ainda são possíveis quando `e` contém apenas um elemento (`L[1]`).

## 8.8 Desempacotamento de parâmetros

Podemos criar funções que recebem um número indeterminado de parâmetros utilizando listas de parâmetros:

```
# Programa 8.14 - Função soma com número indeterminado de parâmetros
def soma(*args):
    s = 0
    for x in args:
        s += x
    return s
soma(1, 2)
soma(2)
soma(5, 6, 7, 8)
soma(9, 10, 20, 30, 40)
```

Também podemos criar funções que combinem parâmetros obrigatórios e uma lista de parâmetros:

```
# Programa 8.15 - Função imprime_maior com número indeterminado de parâmetros
def imprime_maior(mensagem, *numeros):
    maior = None
    for e in numeros:
        if maior is None or maior < e:
            maior = e
    print(mensagem, maior)
imprime_maior("Maior:", 5, 4, 3, 1)
imprime_maior("Max:", *[1, 7, 9])
```

Observe que o primeiro parâmetro é `mensagem`, tornando-o obrigatório. Assim, `imprime_maior()` retorna erro, pois o parâmetro `mensagem` não foi passado, mas `imprime_maior("Max:")` escreve **None**. Isso porque `números` é recebido como uma lista, podendo inclusive estar vazia.

NOTA: para verificarmos se uma variável ou valor é igual a **None**, utilizamos a palavra reservada **is**. Comparar com **None** dessa forma é mais correto, pois **None** é uma instância única (*singleton*) de `NoneType`. Veremos outros exemplos no livro e também no Capítulo 10.

## 8.9 Funções Lambda

Podemos criar funções simples, sem nome, chamadas de funções lambda. Vejamos um exemplo:

```
# Programa 8.16 - Função lambda que recebe um valor e retorna o dobro dele
a = lambda x: x * 2 ❶
print(a(3)) ❷
```

Em ❶, definimos uma função lambda que recebe um parâmetro, no caso `x`, e que retorna o dobro desse número. Observe que tudo foi feito em apenas uma linha. A função é criada e atribuída à variável `a`. Em ❷, utilizamos `a` como uma função normal.

Funções lambda também podem receber mais de um parâmetro, como mostra o Programa 8.17.

```
# Programa 8.17 - Função lambda que recebe mais de um parâmetro
aumento = lambda a, b: (a * b / 100)
aumento(100, 5)
```

Funções lambda são utilizadas quando o código da função é muito simples ou utilizado poucas vezes. Algumas funções da biblioteca padrão do Python também permitem que funções sejam passadas como parâmetro, como o caso do método sort de listas, que recebe um parâmetro key que recebe o elemento e deve retornar a chave a utilizar para a ordenação.

```
>>> L = ["A", "b", "C", "d", "E"]
>>> L.sort()
>>> L
['A', 'C', 'E', 'b', 'd']
>>> L.sort(key=lambda k: k.lower())
>>> L
['A', 'b', 'C', 'd', 'E']
```

## 8.10 Exceções

Exceções são situações inesperadas em nosso código ou apenas algo que não tratamos diretamente no programa. Você já viu vários tipos de exceções até aqui, como `ValueError`, quando tentamos converter uma letra em número, e `IndexError`, quando tentamos acessar o elemento de lista além de seu tamanho. Essas exceções são parte da biblioteca-padrão do Python, mas podemos também criar nossas próprias exceções.

Primeiro, vamos ver como tratar exceções e depois veremos como criar nossas próprias exceções no Capítulo 10.

O bloco responsável por tratar exceções é o **try**:

```
nomes = ["Ana", "Carlos", "Maria"]
for tentativa in range(3):
    try:
        i = int(input("Digite o índice que quer imprimir:"))
        print(nomes[i])
    except ValueError:
        print("Digite um número!")
    except IndexError:
        print("Valor inválido, digite entre -3 e 3")
```

O bloco **try** permite delimitar o código que pode gerar uma exceção. Ele é completado pela declaração **except**. Se nenhum erro ocorrer, o código dentro de **except** nem será executado. Em caso de exceção, dependendo do tipo da exceção, uma

mensagem de erro diferente será exibida. Veja que o **for** nem é interrompido, como aconteceria caso não tivéssemos tratado a exceção.

Exceções são objetos e falaremos mais sobre elas no Capítulo 10. No programa anterior, estamos tratando apenas `ValueError` e `IndexError`. Caso outro tipo de exceção ocorra, ele fará com que nosso programa pare e exiba uma mensagem de erro. Para testar, experimente remover um dos blocos **except**.

Existem vários tipos de exceção no Python. O tipo raiz chama-se **Exception**. Você pode tratar o tipo **Exception** que cobre todos os erros comuns do Python:

```
nomes = ["Ana", "Carlos", "Maria"]
for tentativa in range(3):
    try:
        i = int(input("Digite o índice que quer imprimir:"))
        print(nomes[i])
    except Exception as e:
        print(f"Algo de errado aconteceu: {e}")
```

Veja que agora modificamos a declaração **except** e damos um nome para a exceção, no caso `e`. Vejamos o resultado de alguns testes na tela:

```
Digite o índice que quer imprimir:abc
Algo de errado aconteceu: invalid literal for int() with base 10: 'abc'
Digite o índice que quer imprimir:-100
Algo de errado aconteceu: list index out of range
Digite o índice que quer imprimir:100
Algo de errado aconteceu: list index out of range
```

Veja que agora sabemos apenas que um erro aconteceu. Ainda podemos acessar a informação de tipo (Seção 8.13), mas por enquanto vamos apenas exibir a mensagem de erro da própria exceção.

O mesmo bloco **try** pode conter várias declarações **except**. Além disso, o bloco **try** pode ter uma declaração **finally**. Ao declararmos um bloco com **finally**, estamos dizendo: execute esse bloco, mesmo que aconteça uma exceção, com ou sem tratamento.

Vejamos outra versão do mesmo programa, mas que trata apenas a exceção `ValueError`:

```
nomes = ["Ana", "Carlos", "Maria"]
for tentativa in range(3):
    try:
        i = int(input("Digite o índice que quer imprimir:"))
```

```
        print(nomes[i])
    except ValueError as e:
        print("Digite um número!")
    finally:
        print(f"Tentativa {tentativa + 1}")
```

Vamos testá-lo com alguns valores:

```
Digite o índice que quer imprimir:1
Carlos
Tentativa 1
Digite o índice que quer imprimir:abc
Digite um número!
Tentativa 2
Digite o índice que quer imprimir:100
Tentativa 3
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
IndexError: list index out of range
```

Observe que o **print** foi executado em todos os casos. Quando digitamos 1, não há exceção e o **finally** executa normalmente. Ao digitarmos `abc`, geramos uma exceção `ValueError` que é tratada pelo nosso bloco, imprimindo a mensagem "Digite um número!". Veja que o **finally** também foi executado. No último teste, digitamos o valor 100 que gera uma exceção do tipo `IndexError`. O **finally** é executado (Tentativa 3) e só depois a mensagem de erro com traceback é impressa.

Você pode utilizar um bloco **try** apenas com **finally** ou combinando-o com um ou mais declarações **except**.

Uma exceção não tratada por um bloco **try** é propagada na pilha, da função e bloco atual, a blocos superiores, até que um bloco **try** seja encontrado. Caso não haja bloco **try** para tratar a exceção, a mensagem de erro com traceback que já conhecemos é impressa. Você pode usar blocos **try** para gerenciar exceções geradas dentro de funções e mesmo por funções chamadas por suas funções e assim sucessivamente.

Exceções são um recurso bem popular em várias linguagens de programação. Sua maior vantagem é separar uma execução normal de uma execução anormal. Usando exceções, evitamos retornar e verificar códigos de erro a cada chamada de função, centralizando o tratamento de erros do bloco em um só lugar e facilitando o tratamento de erros.

Podemos também gerar exceções usando a instrução **raise**:

```
nomes = ["Ana", "Carlos", "Maria"]
try:
    i = int(input("Digite o índice que quer imprimir:"))
    print(nomes[i])
except ValueError as e:
    print("Digite um número!")
    raise
finally:
    print(f"Sempre o finally é executado")
```

Ao utilizarmos **raise**, podemos tratar a exceção dentro do **except** e passá-la adiante novamente. Por exemplo:

```
Digite o índice que quer imprimir:abc
Digite um número!
Sempre o finally é executado
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'abc'
```

No exemplo, tanto o **except** quanto o **finally** foram executados. A instrução **raise** fez com que a exceção `ValueError` fosse passada adiante; como não há outro bloco **try** para tratá-la, o interpretador exibe o Traceback com a mensagem de erro.

Vejamos outro exemplo com função:

```
def épar(n):
    try:
        return n % 2
    finally:
        print("Executado antes de retornar")

try:
    print(2, épar(2))
    print("A", épar("A"))
except Exception:
    print("Algo de errado aconteceu")
```

Veja que o **finally** é executado antes de a função retornar e mesmo em caso de exceção:

```
Executado antes de retornar
2 0
Executado antes de retornar
Algo de errado aconteceu
```

Esse exemplo também indica um problema que pode ser evitado: esconder os erros. A mensagem de erro apenas diz que um erro aconteceu. Onde e qual erro foram perdidos.

Utilizando **raise**, podemos também criar nossas próprias exceções:

```
def épar(n):
    try:
        return n % 2
    except Exception:
        raise ValueError("Valor inválido")
```

Que ao executar produz:

```
>>> épar(2)
0
>>> épar([])
Traceback (most recent call last):
  File "<stdin>", line 3, in épar
TypeError: unsupported operand type(s) for %: 'list' and 'int'
During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in épar
ValueError: Valor inválido
```

Veja que a exceção é do tipo `ValueError` e que a mensagem de erro é a que definimos, porém a exceção original também foi reportada e em primeiro lugar. Isso ocorre porque chamamos **raise** dentro de um bloco **except**. Caso esse efeito não seja o desejado, substitua o **raise** por:

```
def épar(n):
    try:
        return n % 2
    except Exception:
        raise ValueError("Valor inválido") from None
```

Que ao executar produz:

```
>>> épar(2)
0
>>> épar([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in épar
ValueError: Valor inválido
```

O bloco **try** também possui uma declaração **else**, assim como **while** e **for**. A declaração **else** é executada apenas se não houver exceção no **try** e pode ser combinada com **except** e **finally**, assim como usada separadamente.

```
while True:
    try:
        v = int(input("Digite um número inteiro (0 sai):"))
        if v == 0:
            break
    except Exception:
        print("Valor inválido! Redigite")
    else:
        print("Parabéns, nenhuma exceção")
    finally:
        print("Executado sempre, mesmo com break")
```

Que executado produz:

```
Digite um número inteiro (0 sai):10
Parabéns, nenhuma exceção
Executado sempre, mesmo com break
Digite um número inteiro (0 sai):abc
Valor inválido! Redigite
Executado sempre, mesmo com break
Digite um número inteiro (0 sai):0
Executado sempre, mesmo com break
```

### Trivia

Sempre que chamamos uma função, a localização (programa, linha) que realizou a chamada é registrada e colocada numa pilha (stack) de chamadas de funções. Quando uma função retorna, esta localização é retirada da pilha e a execução continua normalmente. Como uma função pode chamar outras funções, a pilha de chamadas é um indício muito útil para saber como e onde chamamos nossas funções, principalmente quando estamos procurando um erro ou bug. O traceback que é impresso quando há uma exceção é uma representação do estado da pilha de chamadas: qual função chamou a seguinte, com nome do arquivo e linha do programa. No final do traceback, tempos o tipo da exceção e uma mensagem associada ao erro.

## 8.11 Módulos

Depois de criarmos várias funções, os programas ficaram muito grandes. Precisamos armazenar nossas funções em outros arquivos e, de alguma forma usá-las, sem precisar reescrevê-las, ou pior, copiar e colar.

Python resolve o problema com módulos. Todo arquivo `.py` é um módulo, podendo ser importado com o comando **`import`**.

Vejamos dois programas: `entrada.py` (Programa 8.18) e `soma.py` (Programa 8.19).

```
# Programa 8.18 - Módulo entrada (entrada.py)
def valida_inteiro(mensagem, mínimo, máximo):
    while True:
        try:
            v = int(input(mensagem))
            if v >= mínimo and v <= máximo:
                return v
            else:
                print(f"Digite um valor entre {mínimo} e {máximo}")
        except ValueError:
            print("Você deve digitar um número inteiro")
```

```
# Programa 8.19 - Módulo soma (soma.py) que importa entrada
import entrada
L = []
for x in range(10):
    L.append(entrada.valida_inteiro("Digite um número:", 0, 20))
print(f"Soma: {sum(L)}")
```

Utilizando o comando **`import`** foi possível chamar a função `valida_inteiro`, definida em `entrada.py`. Observe que, para chamar a função `valida_inteiro`, escrevemos o nome do módulo antes do nome da função: `entrada.valida_inteiro`.

Usando módulos, podemos organizar nossas funções em arquivos diferentes e chamá-las quando necessário, sem precisar reescrever tudo. Observe que escrevemos o nome do módulo no **`import`** sem a extensão `.py`.

Nem sempre queremos utilizar o nome do módulo para acessar uma função: `valida_inteiro` pode ser mais interessante que `entrada.valida_inteiro`. Para importar a função `valida_inteiro` de forma a poder chamá-la sem o prefixo do módulo, substitua o **`import`** do Programa 8.19 por **`from`** `entrada` **`import`** `valida_inteiro`.

Depois, modifique o programa substituindo `entrada.valida_inteiro` por apenas `valida_inteiro`.

Você deve utilizar esse recurso com atenção, pois informar o nome do módulo antes da função é muito útil quando os programas crescem, servindo de dica para que se saiba que módulo define tal função, facilitando sua localização e evitando o que se chama de conflito de nomes. Dizemos que um conflito de nomes ocorre quando dois ou mais módulos definem funções com nomes idênticos. Nesse caso, utilizar a notação de chamada `módulo.função` resolve o conflito, pois a combinação do nome do módulo com o nome da função é única, evitando a dúvida de descobrir qual módulo define a função que estamos chamando (quando dois módulos diferentes definem funções com nomes idênticos).

Outra construção que deve ser utilizada com cuidado é `from entrada import *`. Nesse caso, estaríamos importando todas as definições do módulo entrada, e não apenas a função `valida_inteiro`. Essa construção é perigosa, porque, se dois módulos definirem funções com o mesmo nome, a função utilizada no programa será a do último `import`. Esse tipo de erro é especialmente difícil de encontrar em programas grandes, e seu uso não é aconselhado.

## 8.12 Números aleatórios

Uma forma de gerar valores para testar funções e popular listas é utilizar números aleatórios ou randômicos. Um número aleatório pode ser entendido como um número tirado ao acaso, sem qualquer ordem ou sequência predeterminada, como em um sorteio.

Para gerar números aleatórios em Python, vamos utilizar o módulo random. O módulo traz várias funções para geração de números aleatórios e mesmo números gerados com distribuições não uniformes. Se quiser saber mais sobre distribuições, consulte um bom livro de estatística ou a Wikipédia (`http://pt.wikipedia.org/wiki/Distribuição_de_probabilidade`) para matar a curiosidade. Vejamos a função `randint`, que recebe dois parâmetros, sendo o primeiro o início da faixa de valores a considerar para geração; e o segundo, o fim dessa faixa. Tanto o início quanto o fim são incluídos na faixa.

```
import random
for x in range(10):
    print(random.randint(1, 100))
```

Utilizamos `randint` com 1 e 100, logo, os números retornados devem estar na faixa entre 1 e 100. Se chamarmos o número retornado de `x`, teremos `1 <= x <= 100`. Cada vez que executarmos esse programa, teremos valores diferentes sendo gerados.

Essa diferença também é interessante para trabalharmos com jogos ou introduzirmos elementos de sorte em nosso programa.

```
# Programa 8.20 - Adivinhando o número
import random
n = random.randint(1, 10)
x = int(input("Escolha um número entre 1 e 10:"))
if x == n:
    print("Você acertou!")
else:
    print("Você errou.")
```

Veja o Programa 8.20, mesmo lendo a listagem, não podemos determinar a resposta. A cada execução, um número diferente pode ser escolhido. Esse tipo de aleatoriedade é utilizado em vários jogos e torna a experiência única, fazendo com que o mesmo programa possa ser utilizado várias vezes com resultados diferentes.

**Exercício 8.13** Altere o Programa 8.20 de forma que o usuário tenha três chances de acertar o número. O programa termina se o usuário acertar ou errar três vezes.

**TRIVIA**

Python permite que escrevamos $0 < x < 10$ para representar intervalos. Essa forma de escrever uma expressão lógica é mais simples de ler que $x > 0$ and $x < 10$.

Podemos também gerar números aleatórios fracionários ou de ponto flutuante com a função `random`. A função `random` não recebe parâmetros e retorna valores entre 0 e 1.

```
import random
for x in range(10):
    print(random.random())
```

Para obter valores fracionários dentro de uma determinada faixa, podemos usar a função `uniform`:

```
import random
for x in range(10):
    print(random.uniform(15, 25))
```

Podemos utilizar a função `sample` para escolher aleatoriamente elementos de uma lista. Essa função recebe a lista e a quantidade de amostras (*samples*) ou

elementos que queremos retornar. Vejamos como gerar os números de um cartão do jogo da Lotto:

```
import random
print(random.sample(range(1, 101), 6))
```

Se quisermos embaralhar os elementos de uma lista, podemos utilizar a função `shuffle`. Ela recebe a lista a embaralhar, alterando-a:

```
import random
a = list(range(1, 11))
random.shuffle(a)
print(a)
```

**Exercício 8.14** Altere o Programa 7.2, o jogo da forca. Escolha a palavra a adivinhar utilizando números aleatórios.

## 8.13 Função type

A função **type** retorna o tipo de uma variável, função ou objeto em Python. Vejamos alguns exemplos no interpretador:

```
>>> a = 5
>>> type(a)
<class 'int'>
>>> b = "Olá"
>>> type(b)
<class 'str'>
>>> c = False
>>> type(c)
<class 'bool'>
>>> d = 0.5
>>> type(d)
<class 'float'>
>>> f = print
>>> type(f)
<class 'builtin_function_or_method'>
>>> g = []
>>> type(g)
<class 'list'>
```

```
>>> h = {}
>>> type(h)
<class 'dict'>
>>> def função():
...     pass
>>> type(função)
<class 'function'>
```

Observe que o tipo retornado é uma classe. Vejamos como utilizar essa informação em um programa. Para verificar se o tipo retornado pela função **type** equivale a outro tipo, utilizaremos a função **isinstance**, cuja utilização ficará mais clara quando aprendermos mais sobre classes e herança no Capítulo 10.

```
import types
def diz_o_tipo(a):
    tipo = type(a)
    if isinstance(tipo, str):
        return "String"
    elif isinstance(tipo, list):
        return "Lista"
    elif isinstance(tipo, dict):
        return "Dicionário"
    elif isinstance(tipo, int):
        return "Número inteiro"
    elif isinstance(tipo, float):
        return "Número decimal"
    elif isinstance(tipo, types.FunctionType):
        return "Função"
    elif isinstance(tipo, types.BuiltinFunctionType):
        return "Função interna"
    else:
        return str(tipo)
print(diz_o_tipo(10))
print(diz_o_tipo(10.5))
print(diz_o_tipo("Alô"))
print(diz_o_tipo([1, 2, 3]))
print(diz_o_tipo({"a": 1, "b": 50}))
print(diz_o_tipo(print))
print(diz_o_tipo(None))
```

Vamos verificar como utilizar a função **type** para exibir os elementos de uma lista na qual os elementos são de tipos diferentes. Dessa forma, você pode executar operações diferentes dentro da lista, como verificar se um elemento é outra lista ou um dicionário.

```
L = [2, "Alô", ["!"], {"a": 1, "b": 2}]
for e in L:
    print(type(e))
```

Vejamos como navegar em uma lista usando o tipo de seus elementos. Imagine uma lista na qual os elementos podem ser do tipo string ou lista. Se forem do tipo string, podemos simplesmente os imprimir. Se forem do tipo lista, teremos de imprimir cada elemento.

```
# Programa 8.21 - Navegando listas a partir do tipo de seus elementos
import types
L = ["a", ["b", "c", "d"], "e"]
for x in L:
    if isinstance(type(x), str):
        print(x)
    else:
        print("Lista:")
        for z in x:
            print(z)
```

**Exercício 8.15** Utilizando a função **type**, escreva uma função recursiva que imprima os elementos de uma lista. Cada elemento deve ser impresso separadamente, um por linha. Considere o caso de listas dentro de listas, como `L = [1, [2, 3, 4, [5, 6, 7]]]`. A cada nível, imprima a lista mais à direita, como fazemos ao indentar blocos em Python. Dica: envie o nível atual como parâmetro e utilize-o para calcular a quantidade de espaços em branco à esquerda de cada elemento.

## 8.14 List Comprehensions

Uma forma simples de criar listas em Python é utilizando o que chamamos de *list comprehensions*, uma sintaxe capaz de especificar como os elementos de uma lista devem ser gerados. Vejamos:

```
>>> L = [x for x in range(10)]
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

L é uma lista e a expressão [x **for in range**(10)] indica como essa lista será criada. O resultado é uma lista L, contendo números de 0 a 9. Para entender como obtivemos esse resultado, vejamos os componentes da expressão: x **for** x **in range**(10). O primeiro x, antes do **for**, indica o elemento da lista, ou como queremos o elemento na lista. O segundo x, após o **for**, indica o nome do elemento que vamos extrair da expressão após o **in**. E **range**(10) é a função que usamos para gerar elementos. Podemos ler essa expressão como: retorne x como elemento, para todo x de **range**(10). List comprehensions podem ser vistas como fórmulas para criação de listas, em que damos instruções ao interpretador de como criar uma lista com os elementos que desejamos. São muito úteis quando os elementos de uma lista são previsíveis, ou seja, podem ser criados com uma regra. Vejamos o programa equivalente ao exemplo sem utilizar list comprehensions:

```
L = []
for x in range(10):
    L.append(x)
```

Podemos também usar *list comprehensions* com outras listas:

```
>>> Z = [x * 2 for x in [0, 1, 2, 3]]
>>> Z
[0, 2, 4, 6]
```

Nesse exemplo, temos que x * 2 indica como queremos o elemento na nova lista, ou seja, o dobro de cada valor de x. Também usamos a lista [0, 1, 2, 3] como fonte dos elementos a transformar. Essa expressão pode ser lida como: retorne uma lista, em que cada elemento é igual ao dobro do elemento x da lista [0, 1, 2, 3]; ou simplesmente: retorne o dobro de x para cada x em [0, 1, 2, 3].

Podemos usar *list comprehensions* para criar listas mais complexas, por exemplo, nas quais cada elemento é uma tupla:

```
>>> Y = [(x, x * 2) for x in [1, 2, 3]]
>>> Y
[(1, 2), (2, 4), (3, 6)]
```

A mesma sintaxe continua válida, sendo que a parte antes do **for** indica a transformação que faremos nos elementos após o **in**. Podemos ler: faça Y uma lista, em que cada elemento x de [1, 2, 3] é transformado em uma tupla cujo primeiro elemento é o próprio x e o segundo, o dobro de x.

*List comprehensions* também funcionam com outros tipos de dados, como strings:

```
>>> Z = [s.upper() for s in "abcdef"]
>>> Z
['A', 'B', 'C', 'D', 'E', 'F']
```

Nesse caso, estamos usando a string "abcdef" como fonte de nossos elementos. Veja que agora usamos o nome s como nome de cada elemento. Na primeira parte, indicamos que cada elemento s deverá ser transformado pelo resultado de s.upper(). Dessa forma temos uma nova lista, em que cada elemento é uma letra da string, mas convertida para maiúsculas. Podemos ler: faça Z uma nova lista, na qual cada elemento s de "abcdef" deverá ser transformado por s.upper().

Outro recurso de *list comprehensions* é a capacidade de filtrar uma lista utilizando um **if** interno:

```
>>> P = [x for x in range(10) if x % 2 == 0]
>>> P
[0, 2, 4, 6, 8]
```

Podemos ler que P é uma nova lista, em que cada x é igual a ele mesmo (sem transformação) dos elementos de **range(10)**, mas somente os elementos cujo módulo de x e 2 é igual a 0, ou seja, apenas os números pares.

NOTA: nós poderíamos ter gerado os números pares apenas com [x **for** x **in** **range**(0, 10, 2)] ou simplesmente **list**(**range**(0, 10, 2)), que produz o mesmo efeito.

Vejamos outro exemplo, em que inverteremos os elementos de cada tupla dentro de uma lista:

```
>>> D = [(y, x) for x, y in [(4, 3), (1, 2), (8, 9)]]
>>> D
[(3, 4), (2, 1), (9, 8)]
```

As regras do **for** dentro da *list comprehensions* são as mesmas da estrutura de repetição **for** que nos permite desempacotar os elementos de uma lista. No exemplo anterior, desempacotamos cada elemento, chamando o primeiro de x e o segundo de y. Veja também que a transformação indica que os novos elementos estarão no formato (y, x), que realiza a inversão que queríamos.

Um exemplo com desempacotamento mais complexo:

```
>>> G = [(x, y) for *x, y in [(4, 2, 3), (5, 1, 2), (7, 8, 9)]]
>>> G
[([4, 2], 3), ([5, 1], 2), ([7, 8], 9)]
```

Lembrando o desempacotamento de tuplas, temos que *x, y desempacota cada elemento da tupla de forma que x contém todos os elementos da tupla, exceto o último, chamado de y.

```
>>> J = [(x, y) for x, *y in [(4, 2, 3), (5, 1, 2), (7, 8, 9)]]
>>> J
[(4, [2, 3]), (5, [1, 2]), (7, [8, 9])]
```

No caso de J, realizamos o desempacotamento colocando o primeiro elemento em x e os restantes em y.

*List comprehensions* são um conceito poderoso em Python, mas constantemente abusados. Lembre-se de que é muito mais importante manter a clareza do seu código do que utilizar todos os recursos da linguagem em um só programa!

Em nossas regras de transformação, podemos chamar funções, como em:

```
>>> import math
>>> I = [math.sqrt(z) for z in range(0, 10)]
>>> I
[0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.23606797749979,
   2.449489742783178, 2.6457513110645907, 2.8284271247461903, 3.0]
```

ou em:

```
>>> H = [z for z in range(0, 10) if math.sqrt(z) % 1 == 0]
>>> H
[0, 1, 4, 9]
```

Em que H é uma lista cujos elementos têm a raiz quadrada exata (inteira).

## 8.15 Geradores

Até agora, temos falado de geradores e funções geradoras como uma caixa preta. Usamos geradores ao chamarmos a função **range** e **enumerate**, por exemplo. Mas como poderíamos criar nossas próprias funções geradoras? Vejamos o que são iteradores primeiro.

Iteradores são uma forma de acessar os elementos de uma lista ou de outra estrutura de dados, de forma a separar o acesso da representação interna da estrutura em si. Por exemplo, quando percorremos uma lista com **for**, estamos visitando cada elemento, um após o outro, mas sem utilizar índices para acessar cada elemento.

Em Python, podemos chamar a função **iter**, que cria um iterador. O iterador é um objeto que trabalha com um protocolo de acesso bem simples, respondendo a chamadas que retornam o próximo elemento da sequência e geram uma exceção do tipo `StopIteration` quando não há mais elementos a retornar. Vejamos como isso funciona:

```
>>> L = [1, 2, 3]
>>> i = iter(L)
>>> i
<list_iterator object at 0x000001CCA983F240>
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Podemos usar o mesmo raciocínio com dicionários:

```
>>> d = {"A": 1, "B": 2}
>>> i = iter(d)
>>> i
<dict_keyiterator object at 0x000001CCA7A9AA48>
>>> next(i)
'A'
>>> next(i)
'B'
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Observe que os iteradores retornados (objetos) são de tipos diferentes, um é um `list_iterator` e o outro, um `dict_iterator`. Ambos respondem a chamadas da função **`next`**, retornando o próximo elemento e criando uma exceção quando não há mais elementos a retornar. Veja também que, no caso de uma lista, cada iteração com **`next`** retornou um elemento, e, com dicionários, uma chave. Essa é a ideia geral de iteradores, poder navegar ou percorrer uma estrutura de dados, utilizando um protocolo (forma) de acesso único.

Lembrando o que já vimos em capítulos anteriores, essa é exatamente a forma que o **`for`** do Python funciona. Quando escrevemos **`for x in y`**, estamos na realidade trabalhando com **`iter`**`(y)` e, a cada iteração do **`for`**, chamamos **`next`**, atribuindo seu resultado a `x`. O **`for`** também gerencia automaticamente o tratamento da exceção `StopIteration`.

Vários tipos em Python possuem implementações padrões do protocolo de iteração, como: string, listas, dicionários, tuplas e conjuntos.

Como iteradores abstraem a estrutura de dados que percorrem, podemos extrapolar o conceito e criar sequências infinitas, criando elemento por elemento, à medida que precisamos destes. Imagine que a cada chamada de **next** criaríamos um novo elemento. Esse é o princípio de generators ou funções geradoras do Python. Quando criamos uma função geradora em Python, esta, ao ser chamada, cria um iterador. A cada chamada de **next**, a função é chamada novamente e retorna um novo elemento. Vejamos um gerador simples:

```
>>> def gerador_de_numeros():
...     i = 0
...     while True:
...         yield i
...         i += 1
...
>>> g = gerador_de_numeros()
>>> g
<generator object gerador_de_numeros at 0x000001CCA9726048>
>>> next(g)
0
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
4
>>> next(g)
5
>>> next(g)
6
```

O programa gera uma sequência de números infinita. Vamos testá-lo com um **for** que chama `gerador_de_numeros()`:

```
>>> for x in gerador_de_numeros():
...     print(x)
...
```

Como a sequência é infinita, você terá de interromper o programa pressionando Control + C no teclado. Como isso é possível? Como a função se lembra de onde parou? A resposta a essa questão é a instrução **yield**. O **yield** funciona da mesma forma que **return**, mas de maneira diferente quanto a não terminar a função. Você pode entender o **yield** como um **return** que suspende a execução da função, retornando um valor. Ao chamarmos novamente essa função, a execução continua de onde parou da última vez e não a partir da primeira linha da função. Em `gerador_de_numeros`, o valor da variável `i` é preservado entre o **yield** e a continuação da execução ocasionada pela chamada da função **next**.

Vejamos outro tipo de gerador, dessa vez com uma sequência finita:

```
>>> def gerador_fibonacci():
...     p = 0
...     s = 1
...     while s < 10:
...         yield s
...         p, s = s, s + p
...
>>> [x for x in gerador_fibonacci()]
[1, 1, 2, 3, 5, 8]
```

Em `gerador_fibonacci`, criamos um gerador da sequência de Fibonacci, mas introduzimos uma limitação artificial (`s < 10`) para que a sequência gerada fosse finita. Veja que fazemos o **yield** no valor de `s` e que tanto o valor de `p` quanto o de `s` são preservados durante a iteração. Quando o **while** termina (`s == 13`), a função simplesmente retorna.

Geradores têm a vantagem de esconder a lógica necessária para a geração da sequência e podem simplificar enormemente nossos programas. Um exemplo seria um programa que precise trabalhar com a sequência de Fibonacci. Utilizando uma função geradora para criar a sequência, os detalhes da geração da sequência em si ficaram fora do programa principal que trabalha com a sequência gerada. Em casos mais simples, pode-se simplesmente utilizar uma função normal e retornar uma lista ou tupla com a sequência. Geradores são bem interessantes para trabalhar com grandes quantidades de dados, como ao ler um arquivo ou receber dados da rede. Uma aplicação importante é quando nosso programa precisa rodar a cada novo elemento e não pode esperar que todos os elementos sejam gerados, tendo de tratá-los um a um.

## 8.16 Generator Comprehensions

Da mesma forma que podemos usar *list comprehensions*, podemos criar geradores apenas substituindo os colchetes ([]) por parênteses.

```
>>> [x for x in range(10) if x % 3 == 0]
[0, 3, 6, 9]
>>> (x for x in range(10) if x % 3 == 0)
<generator object <genexpr> at 0x000001CCA9726390>
>>> for y in (x for x in range(10) if x % 3 == 0):
...     print(y)
...
0
3
6
9
```

**Exercício 8.16** Escreva um generator capaz de gerar a série dos números primos.

**Exercício 8.17** Escreva um generator capaz de gerar a série de Fibonacci.

# CAPÍTULO 9

# Arquivos

Precisamos de uma forma de armazenar dados permanentemente. Arquivos são uma excelente forma de entrada e saída de dados para programas. Com eles, poderemos ler dados de outros programas e mesmo da internet.

Mas o que é um arquivo? Um arquivo é uma área em disco na qual podemos ler e gravar informações. Essa área é gerenciada pelo sistema operacional do computador, ou seja, não precisamos nos preocupar em como esse espaço é organizado em disco. Do ponto de vista de programas, um arquivo é acessado por nome e é onde podemos ler e escrever linhas de texto ou dados em geral.

Para acessar um arquivo, precisamos abri-lo. Durante a abertura, informamos o nome do arquivo, com o nome do diretório em que ele se encontra (se necessário) e que operações queremos realizar: leitura e/ou escrita. Em Python, abrimos arquivos com a função **open**.

A função **open** utiliza os parâmetros nome e modo. O nome é o nome do arquivo em si, por exemplo, `leiame.txt`. O modo indica as operações que vamos realizar (Tabela 9.1).

*Tabela 9.1 – Modos de abertura de arquivos*

| Modo | Operações |
|---|---|
| `r` | leitura |
| `w` | escrita, apaga o conteúdo se já existir |
| `a` | escrita, mas preserva o conteúdo se já existir |
| `b` | modo binário |
| `+` | atualização (leitura e escrita) |

Os modos podem ser combinados ("`r+`", "`w+`", "`a+`", "`r+b`", "`w+b`", "`a+b`"). As diferenças envolvidas serão discutidas a seguir. A função **open** retorna um objeto do

tipo *file* (arquivo). É esse objeto que vamos utilizar para ler e escrever os dados no arquivo. Utilizamos o método `write` para escrever ou gravar dados no arquivo, `read` para ler e `close` para fechá-lo. Ao trabalharmos com arquivos, devemos sempre realizar o seguinte ciclo: abertura, leitura e/ou escrita, fechamento.

A abertura realiza a ligação entre o programa e o espaço em disco, gerenciado pelo sistema operacional. As etapas de leitura e/ou escrita são as operações que desejamos realizar no programa, e o fechamento informa ao sistema operacional que não vamos mais trabalhar com o arquivo.

O fechamento do arquivo é muito importante, pois cada arquivo aberto consome recursos do computador. Só o fechamento do arquivo garante a liberação desses recursos e preserva a integridade dos dados do arquivo.

Vejamos um exemplo em que vamos escrever o arquivo `números.txt` com 100 linhas. Em cada linha, vamos escrever um número:

```
arquivo = open("números.txt", "w") ❶
for linha in range(1, 101): ❷
        arquivo.write(f"{linha}\n") ❸
arquivo.close() ❹
```

Execute o programa. Se tudo correu bem, nada aparecerá na tela. O programa cria um novo arquivo no diretório atual, ou seja, no mesmo diretório em que você gravou seu programa.

Se você usa Windows, abra o Windows Explorer, escolha a pasta em que grava seus programas e procure o arquivo `números.txt`. Você pode abri-lo com o Notepad ou simplesmente clicando duas vezes em seu nome. Uma vez aberto, observe as linhas: seu programa gerou um arquivo que pode ser aberto por outros programas!

Se você utiliza o Mac OS X, ative o Finder para localizar e abrir seu arquivo. No Linux, você pode utilizar o Nautilus ou simplesmente digitar `less números.txt` na linha de comando.

Em ❶, utilizamos a função **`open`** para abrir o arquivo `números.txt`. Observe que o nome do arquivo é uma string e deve ser escrito entre aspas. O modo escolhido foi `"w"`, indicando escrita ou gravação. O modo `"w"` cria o arquivo se ele não existir. Caso já exista, seu conteúdo é apagado. Para verificar esse efeito, execute o programa novamente e observe que o arquivo continua com 100 linhas. Experimente também escrever algo no arquivo usando o Notepad (ou outro editor de textos simples), grave e execute o programa novamente. Tudo que você escreveu no editor foi perdido, pois o programa apaga o conteúdo do arquivo e começa tudo de novo.

Criamos um **for** ❷ para gerar os números das linhas. Em ❸, escrevemos o número da linha no arquivo, usando o método `write`. Observe que escrevemos `arquivo.write`, pois `write` é um método do objeto arquivo. Observe que escrevemos `f"{linha}\n"` e temos de adicionar o `"\n"` para indicar que queremos passar para uma nova linha.

A linha ❹ fecha o arquivo. O fechamento garante que o sistema operacional foi informado de que não vamos mais trabalhar com o arquivo. Essa comunicação é importante, pois o sistema operacional realiza diversas funções para otimizar a velocidade de suas operações, sendo que uma delas é a de não gravar os dados diretamente no disco, mas em uma memória auxiliar. Quando não fechamos o arquivo, corremos o risco de essa memória auxiliar não ser transferida para o disco e, assim, perdermos o que foi escrito no programa.

Vejamos agora um programa para ler o arquivo e imprimir suas linhas na tela:

```
# Programa 9.1 - Abrindo, lendo e fechando um arquivo
arquivo = open("números.txt", "r") ❶
for linha in arquivo.readlines(): ❷
    print(linha) ❸
arquivo.close() ❹
```

Em ❶, utilizamos a função **open** para abrir o arquivo. O nome é o mesmo utilizado durante a gravação, mas o modo agora é `"r"`, ou seja, leitura. Em ❷ utilizamos o método `readlines`, que gera uma lista em que cada elemento é uma linha do arquivo. Em ❸ simplesmente imprimimos a linha na tela. Em ❹ fechamos o arquivo.

Veja que ❶ é muito parecida com a mesma linha do programa anterior, e que ❹ é idêntica. Isso acontece porque a abertura e o fechamento são partes essenciais na manipulação de arquivos. Sempre que manipularmos arquivos, teremos uma abertura, operações e fechamento.

Até agora estamos trabalhando com arquivos do tipo texto. A característica principal desses arquivos é que seu conteúdo é apenas texto simples, com alguns caracteres especiais de controle. O caractere de controle mais importante em um arquivo-texto é o que marca o fim de uma linha.

No Windows, o fim de linha é marcado por uma sequência de dois caracteres, cujas posições na tabela ASCII são 10 (LF – *Line Feed*, avanço de linha) e 13 (CR – *Carriage Return*, retorno de cursor). No Linux, temos apenas um caractere marcando o fim de linha, o *Line Feed* (10). Já no Mac OS X, temos apenas o *Carriage Return* (13) marcando o fim de uma linha.

Como o fechamento de arquivos é muito importante, Python possui um recurso que garante o fechamento do arquivo, mesmo sem chamarmos close. Isso é importante porque nosso programa pode ser interrompido ou simplesmente parar antes de fecharmos um arquivo. A estrutura **with** permite criarmos um contexto, ou seja, um bloco em que um objeto é válido.

```
# Programa 9.2 - Uso do with
with open("números.txt", "r") as arquivo:
    for linha in arquivo.readlines():
        print(linha)
```

O **with** funciona atribuindo o resultado do **open** à variável arquivo. Compare o Programa 9.1 com o Programa 9.2. Observe que o **for** está dentro do **with**. No fim do bloco **with**, o contexto é destruído. No caso de um contexto de arquivo, este é fechado, como se tivéssemos chamado close manualmente. Dessa forma, garantimos o fechamento do arquivo e evitamos esquecimentos comuns, como de chamar close no final das operações. O uso do **with** também protege o arquivo de exceções. Mesmo que uma exceção ocorra dentro do bloco, o arquivo será fechado.

## 9.1 Parâmetros da linha de comando

Podemos acessar os parâmetros passados ao programa na linha de comando utilizando o módulo sys e trabalhando com a lista argv:

```
import sys
print(f"Número de parâmetros: {len(sys.argv)}")
for n, p in enumerate(sys.argv):
    print(f"Parâmetro {n} = {p}")
```

Experimente chamar o script na linha de comando usando os seguintes parâmetros:

```
fparam.py primeiro segundo terceiro
fparam.py 1 2 3
fparam.py readme.txt 5
```

Observe que cada parâmetro foi passado com um elemento da lista sys.argv e que os parâmetros são separados por espaços em branco na linha de comando, mas que esses espaços são removidos dos elementos em sys.argv. Se você precisar passar um parâmetro com espaços em branco, como um nome formado por várias palavras, escreva-o entre aspas para que ele seja considerado como um só parâmetro.

```
fparam.py "Nome grande" "Segundo nome grande"
```

**Exercício 9.1** Escreva um programa que receba o nome de um arquivo pela linha de comando e que imprima todas as linhas desse arquivo.

**Exercício 9.2** Modifique o programa do Exercício 9.1 para que receba mais dois parâmetros: a linha de início e a de fim para impressão. O programa deve imprimir apenas as linhas entre esses dois valores (incluindo as linhas de início e fim).

## 9.2 Geração de arquivos

Vejamos o Programa 9.3, que gera dois arquivos com 500 linhas cada. O programa distribui os números ímpares e pares em arquivos diferentes.

```
# Programa 9.3 - Gravação de números pares e ímpares em arquivos diferentes
with open("ímpares.txt", "w") as ímpares:
    with open("pares.txt", "w") as pares:
        for n in range(0, 1000):
            if n % 2 == 0:
                pares.write(f"{n}\n")
            else:
                ímpares.write(f"{n}\n")
```

Observe que, para gravar em arquivos diferentes, utilizamos um **if** e dois arquivos abertos para escrita. Utilizando o método `write` dos objetos `pares` e `ímpares`, fizemos a seleção de onde gravar o número. Observe que incluímos o `\n` para indicar fim de linha.

Podemos também escrever esses **with** em um só:

```
# Programa 9.4 - With em uma só linha
with open("ímpares.txt", "w") as ímpares, open("pares.txt", "w") as pares:
    for n in range(0, 1000):
        if n % 2 == 0:
            pares.write(f"{n}\n")
        else:
            ímpares.write(f"{n}\n")
```

## 9.3 Leitura e escrita

Podemos realizar diversas operações com arquivos; entre elas ler, processar e gerar novos arquivos. Utilizando o arquivo `pares.txt` criado pelo Programa 9.4, vejamos como filtrá-lo de forma a gerar um novo arquivo, apenas com números múltiplos de 4:

```
with open("múltiplos de 4.txt", "w") as múltiplos4:
    with open("pares.txt") as pares:
        for l in pares.readlines():
            if int(l) % 4 == 0: ❶
                múltiplos4.write(l)
```

Veja que em ❶ convertemos a linha lida de string para inteiro antes de fazer os cálculos.

**Exercício 9.3** Crie um programa que leia os arquivos `pares.txt` e `ímpares.txt` e que crie um só arquivo `pareseimpares.txt` com todas as linhas dos outros dois arquivos, de forma a preservar a ordem numérica.

**Exercício 9.4** Crie um programa que receba o nome de dois arquivos como parâmetros da linha de comando e que gere um arquivo de saída com as linhas do primeiro e do segundo arquivo.

**Exercício 9.5** Crie um programa que inverta a ordem das linhas do arquivo `pares.txt`. A primeira linha deve conter o maior número; e a última, o menor.

## 9.4 Processamento de um arquivo

Podemos também processar as linhas de um arquivo de entrada com se fossem comandos. Vejamos um exemplo, no qual as linhas cujo primeiro caractere é igual a ; serão ignoradas; as com > serão impressas alinhadas à direita; as com <, alinhadas à esquerda; e as com * serão centralizadas.

Crie um arquivo com as seguintes linhas e salve-o como `entrada.txt`:

```
;Esta linha não deve ser impressa.
>Esta linha deve ser impressa alinhada a direita
*Esta linha deve ser centralizada
Uma linha normal
Outra linha normal
```

Vejamos o programa que o processa:

```
# Programa 9.5 - Processamento de um arquivo
LARGURA = 79
with open("entrada.txt") as entrada:
    for linha in entrada.readlines():
        if linha[0] == ";":
            continue
        elif linha[0] == ">":
            print(linha[1:].rjust(LARGURA))
        elif linha[0] == "*":
            print(linha[1:].center(LARGURA))
        else:
            print(linha)
```

**Exercício 9.6** Modifique o Programa 9.5 para imprimir 40 vezes o símbolo de = se este for o primeiro caractere da linha. Adicione também a opção para parar de imprimir até que se pressione a tecla **Enter** cada vez que uma linha iniciar com . (ponto) como primeiro caractere.

**Exercício 9.7** Crie um programa que leia um arquivo-texto e gere um arquivo de saída paginado. Cada linha não deve conter mais de 76 caracteres. Cada página terá no máximo 60 linhas. Adicione na última linha de cada página o número da página atual e o nome do arquivo original.

**Exercício 9.8** Modifique o programa do Exercício 9.7 para também receber o número de caracteres por linha e o número de linhas por página pela linha de comando.

**Exercício 9.9** Crie um programa que receba uma lista de nomes de arquivo e os imprima, um por um.

**Exercício 9.10** Crie um programa que receba uma lista de nomes de arquivo e que gere apenas um grande arquivo de saída.

**Exercício 9.11** Crie um programa que leia um arquivo e crie um dicionário em que cada chave é uma palavra e cada valor é o número de ocorrências no arquivo.

**Exercício 9.12** Modifique o programa do Exercício 9.11 para também registrar a linha e a coluna de cada ocorrência da palavra no arquivo. Para isso, utilize listas nos valores de cada palavra, guardando a linha e a coluna de cada ocorrência.

**Exercício 9.13** Crie um programa que imprima as linhas de um arquivo. Esse programa deve receber três parâmetros pela linha de comando: o nome do arquivo, a linha inicial e a última linha a imprimir.

**Exercício 9.14** Crie um programa que leia um arquivo-texto e elimine os espaços repetidos entre as palavras e no fim das linhas. O arquivo de saída também não deve ter mais de uma linha em branco repetida.

**Exercício 9.15** Altere o Programa 7.2, o jogo da forca. Utilize um arquivo em que uma palavra seja gravada a cada linha. Use um editor de textos para gerar o arquivo. Ao iniciar o programa, utilize esse arquivo para carregar (ler) a lista de palavras. Experimente também perguntar o nome do jogador e gerar um arquivo com o número de acertos dos cinco melhores.

Usando arquivos, podemos gravar dados de forma a reutilizá-los nos programas. Até agora, tudo que inserimos ou digitamos nos programas era perdido no fim da execução. Com arquivos, podemos registrar essa informação de forma mais duradoura e reutilizá-la. Arquivos podem ser utilizados para fornecer uma grande quantidade de dados aos programas. Vejamos um exemplo no qual gravaremos nomes e telefones em um arquivo-texto. Utilizaremos um menu pra deixar o usuário decidir quando ler o arquivo e quando gravá-lo.

Execute o Programa 9.6 e analise-o cuidadosamente.

```
# Programa 9.6 - Controle de uma agenda de telefones
agenda = []
def pede_nome():
    return input("Nome: ")
def pede_telefone():
    return input("Telefone: ")
def mostra_dados(nome, telefone):
    print(f"Nome: {nome} Telefone: {telefone}")
def pede_nome_arquivo():
    return input("Nome do arquivo: ")
def pesquisa(nome):
    mnome = nome.lower()
    for p, e in enumerate(agenda):
        if e[0].lower() == mnome:
            return p
    return None
def novo():
    global agenda
    nome = pede_nome()
    telefone = pede_telefone()
    agenda.append([nome, telefone])
```

```
def apaga():
    global agenda
    nome = pede_nome()
    p = pesquisa(nome)
    if p is not None:
        del agenda[p]
    else:
        print("Nome não encontrado.")
def altera():
    p = pesquisa(pede_nome())
    if p is not None:
        nome = agenda[p][0]
        telefone = agenda[p][1]
        print("Encontrado:")
        mostra_dados(nome, telefone)
        nome = pede_nome()
        telefone = pede_telefone()
        agenda[p] = [nome, telefone]
    else:
        print("Nome não encontrado.")
def lista():
    print("\nAgenda\n\n------")
    for e in agenda:
        mostra_dados(e[0], e[1])
    print("------\n")
def lê():
    global agenda
    nome_arquivo = pede_nome_arquivo()
    with open(nome_arquivo, "r", encoding="utf-8") as arquivo:
        agenda = []
        for l in arquivo.readlines():
            nome, telefone = l.strip().split("#")
            agenda.append([nome, telefone])
def grava():
    nome_arquivo = pede_nome_arquivo()
    with open(nome_arquivo, "w", encoding="utf-8") as arquivo:
        for e in agenda:
            arquivo.write(f"{e[0]}#{e[1]}\n")
```

```
def valida_faixa_inteiro(pergunta, início, fim):
    while True:
        try:
            valor = int(input(pergunta))
            if início <= valor <=fim:
                return valor
        except ValueError:
            print(f"Valor inválido, favor digitar entre {início} e {fim}")
def menu():
    print("""
  1 - Novo
  2 - Altera
  3 - Apaga
  4 - Lista
  5 - Grava
  6 - Lê

  0 - Sai
""")
    return valida_faixa_inteiro("Escolha uma opção: ", 0, 6)
while True:
    opção = menu()
    if opção == 0:
        break
    elif opção == 1:
        novo()
    elif opção == 2:
        altera()
    elif opção == 3:
        apaga()
    elif opção == 4:
        lista()
    elif opção == 5:
        grava()
    elif opção == 6:
        lê()
```

Os exercícios a seguir modificam o Programa 9.6.

**Exercício 9.16** Explique como os campos nome e telefone são armazenados no arquivo de saída.

**Exercício 9.17** Altere o Programa 9.6 para exibir o tamanho da agenda no menu principal.

**Exercício 9.18** O que acontece se nome ou telefone contiverem o caractere usado como separador em seus conteúdos? Explique o problema e proponha uma solução.

**Exercício 9.19** Altere a função lista para que exiba também a posição de cada elemento.

**Exercício 9.20** Adicione a opção de ordenar a lista por nome no menu principal.

**Exercício 9.21** Nas funções de `altera` e `apaga`, peça que o usuário confirme a alteração e exclusão do nome antes de realizar a operação em si.

**Exercício 9.22** Ao ler ou gravar uma nova lista, verifique se a agenda atual já foi gravada. Você pode usar uma variável para controlar quando a lista foi alterada (novo, altera, apaga) e reinicializar esse valor quando ela for lida ou gravada.

**Exercício 9.23** Altere o programa para ler a última agenda lida ou gravada ao inicializar. Dica: utilize outro arquivo para armazenar o nome.

**Exercício 9.24** O que acontece com a agenda se ocorrer um erro de leitura ou gravação? Explique.

**Exercício 9.25** Altere as funções `pede_nome` e `pede_telefone` de forma a receberem um parâmetro opcional. Caso esse parâmetro seja passado, utilize-o como retorno caso a entrada de dados seja vazia.

**Exercício 9.26** Altere o programa de forma a verificar a repetição de nomes. Gere uma mensagem de erro caso duas entradas na agenda tenham o mesmo nome.

**Exercício 9.27** Modifique o programa para também controlar a data de aniversário e o email de cada pessoa.

**Exercício 9.28** Modifique o programa de forma a poder registrar vários telefones para a mesma pessoa. Permita também cadastrar o tipo de telefone: celular, fixo, residência ou trabalho.

## 9.5 Geração de HTML

Páginas web nada mais são que a combinação de textos e imagens, interligadas por links. Hoje essa definição vai além com a crescente popularidade de vídeo e as chamadas Aplicações de Internet Ricas (*Rich Internet Applications*),

normalmente escritas em Javascript. Como aqui tratamos apenas de introduzir conceitos de programação, vejamos como utilizar o que já sabemos sobre arquivos para gerar *home pages* ou páginas web simples.

Toda página web é escrita em uma linguagem de marcação chamada HTML (*Hypertext Mark-up Language* – Linguagem de Marcação de Hipertexto). Primeiro, precisamos entender o que são marcações. Como o formato HTML é definido apenas com texto simples, ou seja, sem caracteres especiais de controle, ele utiliza marcações, que são sequências especiais de texto delimitado pelos caracteres de menor (<) e maior (>). Essas sequências são chamadas de tags e podem iniciar ou finalizar um elemento. O elemento de mais alto nível de um documento HTML é chamado de `<html>`. Escreveremos nossas páginas web entre as tags `<html>` e `</html>`, em que a primeira marca o início do documento; e a segunda, seu fim. Diremos que `<html>` é a *tag* que inicia o elemento `html`, e que a tag `</html>` é a tag que o finaliza. Vamos seguir os conselhos definidos na especificação do padrão HTML 5 e definir a página inicial como mostrado a seguir. Observe que à esquerda de cada linha apresentamos seu número. Você não deve digitar esse número em seu arquivo.

Página web simples `ola.html`:

```
 1 <!DOCTYPE html>
 2 <html lang="pt-BR">
 3 <head>
 4 <meta charset="utf-8">
 5 <title>Título da Página</title>
 6 </head>
 7 <body>
 8 Olá!
 9 </body>
10 </html>
```

Você pode escrever essa página usando um editor de textos. Grave o arquivo com o nome de `ola.html`. Abra em seu browser (navegador) de internet preferido: FireFox, Chrome, Safari, Edge etc. A forma mais fácil de abrir um arquivo em disco no browser é selecionar o menu **Arquivo** e depois a opção **Abrir**. Escolha o arquivo *ola.html* e veja o que aparece na tela. Se os acentos não aparecerem corretamente, verifique se você gravou o arquivo como texto UTF-8, da mesma forma que fazemos com os programas.

Vamos analisar essa página linha por linha. A linha 1 faz parte do formato e deve sempre ser incluída: ela indica que o documento foi escrito no formato HTML.

A linha 2 contém a *tag* `html`, mas não apenas o nome da tag e o atributo `lang`, cujo valor é `pt-BR`. Observe que o início da tag é marcado com < e o fim com > e que `html lang="pt-BR"` foi escrito na mesma tag, ou seja, entre os símbolos de < e >. O atributo `lang` indica a língua utilizada ao escrever o documento: o valor `pt-BR` indica português do Brasil.

A linha 4 especifica que o arquivo utiliza o formato de codificação de caracteres UTF-8 que permite escrever em português sem problemas. No entanto, você deve garantir que realmente escreveu o arquivo usando UTF-8. No Windows, você pode usar o editor Sublime ou Notepad++ para gerar arquivos UTF-8; no Linux e no Mac OS X, esse tipo de codificação é usado por padrão e não deve causar problemas. Um sinal de problema de codificação é se você não conseguir ler `Olá` na tela ou se o título da página não aparecer corretamente. Se alterar o arquivo HTML, você deve solicitar ao browser que recarregue (*reload*) a página para visualizar as mudanças.

A linha 5 traz o título da página. Observe que o título é simplesmente escrito entre `<title>` e `</title>`. Observe também que escrevemos `title` e `meta` dentro do elemento `head` que começa na linha 3 e termina na linha 6. O formato HTML especifica que elementos devem ser escritos e onde podemos escrevê-los. Para mais detalhes, você pode consultar a especificação do formato HTML 5 na internet. Por enquanto, considere que nosso primeiro arquivo html será utilizado como um modelo em nossos programas.

Entre as linhas 7 e 9, definimos o elemento `body`, que é o corpo da página web. A linha 8 contém apenas a mensagem Olá!. Finalmente, na linha 10, temos a tag que finaliza o elemento `html`, ou seja, `</html>`, marcando o fim do documento. Experimente alterar esse arquivo, escrevendo uma nova mensagem na linha 8. Experimente também digitar várias linhas de texto (você pode copiar e colar um texto de 10 ou 12 linhas). Não se esqueça de salvar o arquivo com as modificações e de recarregar a página no browser para visualizá-las.

Como páginas web são arquivos texto, podemos criá-las facilmente em Python. Veja o programa a seguir que cria uma página HTML.

```
# Programa 9.7 - Criação de uma página inicial em Python
with open("página.html", "w", encoding="utf-8") as página:
    página.write("<!DOCTYPE html>\n")
    página.write("<html lang=\"pt-BR\">\n")
    página.write("<head>\n")
    página.write("<meta charset=\"utf-8\">\n")
    página.write("<title>Título da Página</title>\n")
```

```
página.write("</head>\n")
página.write("<body>\n")
página.write("Olá!")
for l in range(10):
    página.write(f"<p>{l}</p>\n")
página.write("</body>\n")
página.write("</html>\n")
```

Simplesmente escrevemos nossa página dentro de um programa. Observe que ao escrevermos aspas dentro de aspas, como na tag `html`, em que o valor de `lang` é `utf-8` e é escrito entre aspas, tivemos o cuidado de colocar uma barra antes das aspas, informando, dessa forma, que as aspas não fazem parte do programa e que não marcam o fim da string. Um detalhe muito importante é o parâmetro extra que passamos em **open**. O parâmetro `encoding="utf-8"` informa que queremos os arquivos com a codificação UTF-8. Essa codificação tem de ser a mesma declarada no arquivo html, caso contrário, teremos problemas com caracteres acentuados.

Execute o programa e abra o arquivo *página.html* em seu browser. Modifique o programa para gerar 100 parágrafos em vez de 10. Execute-o novamente e veja o resultado no browser (abrindo o arquivo de novo ou clicando em recarregar).

Em nossos primeiros testes, vimos que escrevendo várias linhas de texto no corpo da página não alteramos o formato de saída da página. Isso porque o formato html ignora espaços em branco repetidos e quebras de linha. Se quisermos mostrar o texto em vários parágrafos, devemos utilizar o elemento `p`. Assim, as tags `<p>` e `</p>` marcam o início e o fim de um parágrafo, como no Programa 9.7.

Python oferece recursos mais interessantes para trabalhar com strings, como aspas triplas que permitem escrever longos textos mais facilmente. Elas funcionam como as aspas, mas permitem digitar a mensagem em várias linhas. Vejamos no programa o uso de aspas triplas a seguir:

```
with open("página.html", "w", encoding="utf-8") as página:
    página.write("""
<!DOCTYPE html>
<html lang="pt-BR">
<head>
<meta charset="utf-8">
<title>Título da Página</title>
</head>
<body>
Olá!
""")
```

```
    for l in range(10):
        página.write(f"<p>{l}</p>\n")
    página.write("""
</body>
</html>
""")
```

Outra vantagem é que não precisamos colocar uma barra antes das aspas, pois agora o fim das aspas também é triplo (“”””), não sendo mais necessário diferenciá-lo. Experimente agora modificar o programa retirando o \n do final da linha dentro do **for**. Execute-o e veja o resultado no browser. Você deve perceber que, mesmo sem utilizar quebras de linhas no arquivo, o resultado permaneceu o mesmo, mas que, em html, todos os parágrafos foram escritos em uma só linha. Isso porque espaços adicionais (repetidos) e quebras de linha são ignorados (ou quase ignorados) em HTML. No entanto, observe que a página criada é mais difícil de ler para nós.

O formato HTML contém inúmeras tags que são continuamente revisadas e expandidas. Além das tags que já conhecemos, temos também as que marcam os cabeçalhos dos documentos: `h1`, `h2`, `h3`, `h4`, `h5` e `h6`. Os números de 1 a 6 são utilizados para indicar o nível da seção ou subseção do documento, como faríamos no Microsoft Word ou OpenOffice com estilos de cabeçalho.

Vejamos como gerar páginas web a partir de um dicionário. Vamos utilizar as chaves como título das seções, e o valor como conteúdo do parágrafo (Programa 9.8).

```
# Programa 9.8 - Geração de uma página web a partir de um dicionário
filmes = {
    "drama": ["Cidadão Kane", "O Poderoso Chefão"],
    "comédia": ["Tempos Modernos", "American Pie", "Dr. Dolittle"],
    "policial": ["Chuva Negra", "Desejo de Matar", "Difícil de Matar"],
    "guerra": ["Rambo", "Platoon", "Tora!Tora!Tora!"]
}
with open("filmes.html", "w", encoding="utf-8") as página:
    página.write("""
<!DOCTYPE html>
<html lang="pt-BR">
<head>
<meta charset="utf-8">
<title>Filmes</title>
</head>
<body>
""")
```

```
for c, v in filmes.items():
    página.write(f"<h1>{c}</h1>\n")
    for e in v:
        página.write(f"<h2>{e}</h2>\n")
página.write("</body></html>")
```

**Exercício 9.29** Modifique o Programa 9.8 para utilizar o elemento `p` em vez de `h2` nos filmes.

**Exercício 9.30** Modifique o Programa 9.8 para gerar uma lista html, usando os elementos `ul` e `li`. Todos os elementos da lista devem estar dentro do elemento `ul`, e cada item dentro de um elemento `li`. Exemplo:

```
<ul><li>Item1</li><li>Item2</li><li>Item3</li></ul>V
```

Você pode ler mais sobre HTML e CSS (*Cascading Style-Sheets*) para gerar páginas profissionais em Python.

## 9.6 Arquivos e diretórios

Agora que já sabemos ler, criar e alterar arquivos, vamos aprender como listá-los, manipular diretórios, verificar o tamanho e a data de criação de arquivos em disco.

Para começar, precisamos que os programas saibam de onde estão sendo executados, ou seja, qual é o diretório corrente. Vamos utilizar a função `getcwd` do módulo `os` para obter esse valor:

```
>>> import os
>>> os.getcwd()
```

Também podemos trocar de diretório em Python, mas, antes, precisamos criar alguns diretórios para teste. Na linha de comando, digite:

```
mkdir a
mkdir b
mkdir c
```

Se preferir, você pode criar as pastas `a`, `b` e `c` usando o Windows Explorer, o Finder ou outro utilitário de sua preferência. Agora, vejamos como trocar de diretório em Python:

```
import os
os.chdir("a")
print(os.getcwd())
```

```
os.chdir("..")
print(os.getcwd())
os.chdir("b")
print(os.getcwd())
os.chdir("../c")
print(os.getcwd())
```

A função `chdir` muda o diretório atual, por isso a função `getcwd` apresenta valores diferentes. Você pode referenciar um arquivo apenas usando seu nome se ele estiver no diretório atual ou de trabalho. O que `chdir` faz é mudar o diretório de trabalho, permitindo que você acesse seus arquivos mais facilmente.

Quando passamos "`..`" para `chdir` ou qualquer outra função que manipule arquivos e diretórios, estamos nos referindo ao diretório pai ou de nível superior. Por exemplo, considere o diretório `z`, que contém os diretórios `h`, `i`, `j`. Dentro do diretório (ou pasta) `j`, temos o diretório `k` (Figura 9.1). Quando um diretório está dentro de outro, dizemos que o diretório pai contém o diretório filho. No exemplo, `k` é filho de `j`, e `j` é pai de `k`. Temos também que `h`, `i`, `j` são todos filhos de `z`, ou seja, `z` é o pai de `h`, `i`, `j`.

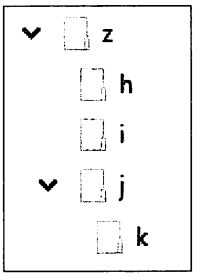


*Figura 9.1 – Estrutura de diretórios no Windows Explorer do Windows 10.*

Esse tipo de endereçamento é sempre relativo ao diretório corrente, por isso é tão importante saber em que diretório estamos. Se o diretório corrente for `k`, `..` refere-se a `j`. Mas, se o diretório atual for `i`, `..` refere-se a `z`. Podemos também combinar vários `..` no mesmo endereço ou caminho (*path*). Por exemplo, se o diretório atual for `k`, `../..` será uma referência a `z` e assim por diante. Também podemos usar o diretório pai para navegar entre os diretórios. Por exemplo, para acessar `h`, estando em `k`, podemos escrever `../../h`.

Podemos também criar diretórios nos programas utilizando a função `mkdir`:

```
import os
os.mkdir("d")
os.mkdir("e")
```

```
os.mkdir("f")
print(os.getcwd())
os.chdir("d")
print(os.getcwd())
os.chdir("../e")
print(os.getcwd())
os.chdir("..")
print(os.getcwd())
os.chdir("f")
print(os.getcwd())
```

A função `mkdir` cria apenas um diretório de cada vez. Se precisar criar um diretório, sabendo ou não se os superiores foram criados, use a função `makedirs`, que cria todos os diretórios intermediários de uma só vez:

```
import os
os.makedirs("avô/pai/filho")
os.makedirs("avô/mãe/filha")
```

Abra o diretório atual usando o Windows Explorer ou Finder, no Mac OS X, para ver os diretórios criados (Figura 9.2).

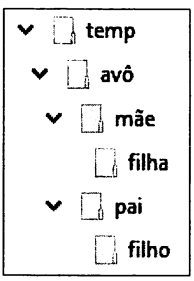


*Figura 9.2 – Estrutura de diretórios criada em disco no Windows Explorer do Windows 10.*

Você pode mudar o nome de um diretório ou arquivo, ou seja, renomeá-lo usando a função `rename`:

```
import os
os.mkdir("velho")
os.rename("velho", "novo")
```

A função `rename` também pode ser utilizada para mover arquivos, bastando especificar o mesmo nome em outro diretório:

```
import os
os.makedirs("avô/pai/filho")
os.makedirs("avô/mãe/filha")
os.rename("avô/pai/filho", "avô/mãe/filho")
```

Se você quiser apagar um diretório, utilize a função `rmdir`. Se quiser apagar um arquivo, use a função `remove`:

```
import os
# Cria um arquivo e o fecha imediatamente
open("morimbundo.txt", "w").close()
os.mkdir("vago")
os.rmdir("vago")
os.remove("morimbundo.txt")
```

Podemos também solicitar uma listagem de todos os arquivos e diretórios usando a função `listdir`. Os arquivos e diretórios serão retornados como elementos da lista:

```
import os
print(os.listdir("."))
print(os.listdir("avô"))
 print(os.listdir("avô/pai"))
print(os.listdir("avô/mãe"))
```

Em que . significa o diretório atual. A lista contém apenas o nome de cada arquivo. Vamos ver como obter o tamanho do arquivo e as datas de criação, acesso e modificação a seguir com o módulo `os.path`.

O módulo `os.path` traz várias outras funções que vamos utilizar para obter mais informações sobre os arquivos em disco. As duas primeiras são `isdir` e `isfile`, que retornam **`True`** se o nome passado for um diretório ou um arquivo respectivamente:

```
import os
import os.path
for a in os.listdir("."):
    if os.path.isdir(a):
        print(f"{a}/")
    elif os.path.isfile(a):
        print(a)
```

Execute o programa e veja que imprimimos apenas os nomes de diretórios e arquivos, sendo que adicionamos uma barra no final dos nomes de diretório.

Podemos também verificar se um diretório ou arquivo já existe com a função exists:

```
# Programa 9.9 - Verificação se um diretório ou arquivo já existe
import os.path
if os.path.exists("z"):
    print("O diretório z existe.")
else:
    print("O diretório z não existe.")
```

**Exercício 9.31** Crie um programa que corrija o Programa 9.9 de forma a verificar se z existe e é um diretório.

**Exercício 9.32** Modifique o Programa 9.9 de forma a receber o nome do arquivo ou diretório a verificar pela linha de comando. Imprima se existir e se for um arquivo ou um diretório.

**Exercício 9.33** Crie um programa que gere uma página html com links para todos os arquivos jpg e png encontrados a partir de um diretório informado na linha de comando.

Temos também outras funções que retornam mais informações sobre arquivos e diretórios como seu tamanho e datas de modificação, criação e acesso:

```
import os
import os.path
import time
import sys
nome = sys.argv[1]
print(f"Nome: {nome}")
print(f"Tamanho: {os.path.getsize(nome)}")
print(f"Criado: {time.ctime(os.path.getctime(nome))}")
print(f"Modificado: {time.ctime(os.path.getmtime(nome))}")
print(f"Acessado: {time.ctime(os.path.getatime(nome))}")
```

Em que `getsize` retorna o tamanho do arquivo em bytes, `getctime` retorna a data e hora de criação, `getmtime` de modificação e `getatime` de acesso. Observe que chamamos `time.ctime` para transformar a data e hora retornadas por `getmtime`, `getatime` e `getctime` em string. Isso é necessário porque o valor retornado é expresso em segundos e precisa ser corretamente convertido para ser exibido.

## 9.7 Um pouco sobre o tempo

O módulo `time` traz várias funções para manipular o tempo. Uma delas foi apresentada na seção anterior: `time.ctime`, que converte um valor em segundos após 01/01/1970 em string. Temos também a função `gmtime`, que retorna uma tupla com componentes do tempo separados em elementos. Vejamos alguns exemplos no interpretador:

```
>>> import time
>>> agora = time.time()
>>> agora
1277310220.906508
>>> time.ctime(agora)
'Wed Jun 23 18:23:40 2010'
>>> agora2 = time.localtime()
>>> agora2
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=23, tm_hour=18, tm_min=23,
   tm_sec=40, tm_wday=2, tm_yday=174, tm_isdst=1)
>>> time.gmtime(agora)
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=23, tm_hour=16, tm_min=23,
   tm_sec=40, tm_wday=2, tm_yday=174, tm_isdst=0)
```

A função `time.time` retorna a hora atual em segundos, usando o horário de Greenwich ou UTC (Tempo Universal Coordenado). No exemplo anterior, atribuímos seu resultado à variável `agora` e convertemos em string usando `time.ctime`. Se você deseja trabalhar com a hora em seu fuso horário, utilize `time.localtime`, como mostrado com a variável `agora2`. Observe que `time.localtime` retornou uma tupla e que `time.time` retorna apenas um número. Podemos utilizar a função `gmtime` para converter a variável `agora` em uma tupla de nove elementos, como a retornada por `time.localtime` (Tabela 9.2).

*Tabela 9.2 – Elementos da tupla de tempo retornada por gmtime*

| Posição | Nome | Descrição |
|---|---|---|
| 0 | tm_year | ano |
| 1 | tm_mon | mês |
| 2 | tm_mday | dia |
| 3 | tm_hour | hora |
| 4 | tm_min | minutos |
| 5 | tm_sec | segundos |

| Posição | Nome | Descrição |
|---|---|---|
| 6 | tm_wday | dia da semana entre 0 e 6, em que segunda-feira é 0 |
| 7 | tm_yday | dia do ano, varia de 1 a 366. |
| 8 | tm_isdst | horário de verão, em que 1 indica estar no horário de verão. |

Podemos também acessar esses dados por nome:

```
import time
agora = time.localtime()
print(f"Ano: {agora.tm_year}")
print(f"Mês: {agora.tm_mon}")
print(f"Dia: {agora.tm_mday}")
print(f"Hora: {agora.tm_hour}")
print(f"Minuto: {agora.tm_min}")
print(f"Segundo: {agora.tm_sec}")
print(f"Dia da semana: {agora.tm_wday}")
print(f"Dia no ano: {agora.tm_yday}")
print(f"Horário de verão: {agora.tm_isdst}")
```

A função `time.strftime` permite a formatação do tempo em string. Você pode passar o formato desejado para a string, seguindo os códigos de formatação da Tabela 9.3.

*Tabela 9.3 – Códigos de formatação de strftime*

| Código | Descrição |
|---|---|
| %a | dia da semana abreviado |
| %A | nome do dia da semana |
| %b | nome do mês abreviado |
| %B | nome do mês completo |
| %c | data e hora conforme configuração regional |
| %d | dia do mês (01-31) |
| %H | hora no formato 24 h (00-23) |
| %I | hora no formato 12 h |
| %j | dia do ano 001-366 |
| %m | mês (01-12) |
| %M | minutos (00-59) |
| %p | AM ou PM |
| %S | segundos (00-61) |

| Código | Descrição |
|---|---|
| %U | número da semana (00-53), em que a semana 1 começa após o primeiro domingo. |
| %w | dia da semana (0-6), em que 0 é o domingo |
| %W | número da semana (00-53), em que a semana 1 começa após a primeira segunda-feira |
| %x | representação regional da data |
| %X | representação regional da hora |
| %y | ano (00-99) |
| %Y | ano com 4 dígitos |
| %Z | nome do fuso horário |
| %% | símbolo de % |

Se precisar converter uma tupla em segundos, utilize a função `timegm` do módulo `calendar`. Se precisar trabalhar com data e hora em seus programas, consulte a documentação do Python sobre os módulos `time`, `datetime`, `calendar` e `locale`.

**Exercício 9.34** Altere o Programa 7.2, o jogo da forca. Dessa vez, utilize as funções de tempo para cronometrar a duração das partidas.

## 9.8 Uso de caminhos

Uma tarefa comum quando se trabalha com arquivos é manipular caminhos. Como essa tarefa depende do sistema operacional – e cada sistema tem suas próprias características, como "/" no Linux e no Mac OS X para separar o nome dos diretórios e "\" no Windows –, a biblioteca padrão do Python traz algumas funções interessantes. Aqui veremos as mais importantes: a documentação do Python traz a lista completa. Vejamos alguns exemplos dessas funções quando chamadas no interpretador Python:

```
>>> import os.path
>>> caminho = "i/j/k"
>>> os.path.abspath(caminho)
'C:\\Python37\\i\\j\\k'
>>> os.path.basename(caminho)
'k'
>>> os.path.dirname(caminho)
'i/j'
```

```
>>> os.path.split(caminho)
('i/j', 'k')
>>> os.path.splitext("arquivo.txt")
('arquivo', '.txt')
>>> os.path.splitdrive("c:/Windows")
('c:', '/Windows')
```

A função `abspath` retorna o caminho absoluto do path passado como parâmetro. Se o caminho não começar com `/`, o diretório atual é acrescentado, retornando o caminho completo a partir da raiz. No caso do Windows, incluindo também a letra do disco (*drive*), no caso `C:`.

Observe que o caminho "`j/k`" é apenas uma string. As funções de `os.path` não verificam se esse caminho realmente existe. Na realidade, `os.path`, na maioria das vezes, oferece apenas funções inteligentes para manipulação de caminhos como strings.

A função `basename` retorna apenas a última parte do caminho, no exemplo, `k`. Já a função `dirname` retorna o caminho à esquerda da última barra. Mais uma vez, essas funções não verificam se `k` é um arquivo ou diretório. Considere que `basename` retorna a parte do caminho à direita da última barra, e que `dirname` retorna o caminho à esquerda. Você até pode combinar o resultado dessas duas funções com a função `split`, que retorna uma tupla em que os elementos são iguais aos resultados de `dirname` e `basename`.

No Windows, você pode também usar a função `splitdrive` para separar a letra do drive do caminho em si. A função retorna uma tupla, em que a letra do drive é o primeiro elemento; e o restante do caminho, o segundo.

A função `join` junta os componentes de um caminho, separando-os com barras, se necessário. No Windows, a função verifica se o nome termina com "`:`" e, nesse caso, não insere uma barra, permitindo a criação de um caminho relativo. Podemos combinar o resultado dessa função com `abspath` e obter um caminho a partir da raiz. Veja que a manipulação da letra do drive é feita automaticamente. Por exemplo:

```
>>> import os.path
>>> os.path.join("c:", "dados", "programas")
'c:dados\\programas'
>>> os.path.abspath(os.path.join("c:", "dados", "programas"))
'C:\\Python37\\dados\\programas'
```

## 9.9 Visita a todos os subdiretórios recursivamente

A função `os.walk` facilita a navegação em uma árvore de diretórios. Imagine que deseje percorrer todos os diretórios a partir de um diretório inicial, retornando o nome do diretório sendo visitado (`raiz`), os diretórios encontrados dentro do diretório sendo visitado (`diretórios`) e uma lista de seus arquivos (`arquivos`). Observe e execute o Programa 9.10. Você deve executá-lo passando o diretório inicial a visitar na linha de comando.

```
# Programa 9.10 - Árvore de diretórios sendo percorrida
import os
import sys
for raiz, diretorios, arquivos in os.walk(sys.argv[1]):
    print("\nCaminho:", raiz)
    for d in diretorios:
        print(f"  {d}/")
    for f in arquivos:
        print(f"  {f}")
    print(f"{len(diretorios)} diretório(s), {len(arquivos)} arquivo(s)")
```

A grande vantagem da função `os.walk` é que ela visita automaticamente todos os subdiretórios dentro do diretório passado como parâmetro, fazendo-o repetidamente até navegar a árvore de diretórios completa.

Combinando a função `os.walk` às funções de manipulação de diretórios e arquivos que já conhecemos, você pode escrever programas para manipular árvores de diretórios completas.

**Exercício 9.35** Utilizando a função `os.walk`, crie uma página HTML com o nome e tamanho de cada arquivo de um diretório passado e de seus subdiretórios.

**Exercício 9.36** Utilizando a função `os.walk`, crie um programa que calcule o espaço ocupado por cada diretório e subdiretório, gerando uma página html com os resultados.

## 9.10 Data e hora

Python possui os tipos `datetime` e `time` no módulo também chamado `datetime`. Esse módulo define uma série de operações para auxiliar a manipulação de datas e horas. O tipo `datetime` representa uma data e uma hora, o tipo `date` apenas uma data e `time` apenas a hora.

```
>>> import datetime
>>> datetime.date.today()
datetime.date(2018, 11, 1)
>>> datetime.datetime.now()
datetime.datetime(2018, 11, 1, 13, 45, 11, 877875)
>>> datetime.datetime.now().time()
datetime.time(13, 45, 26, 437575)
```

Podemos também acessar o dia, o mês e o ano como propriedades da data:

```
>>> data = datetime.datetime.today()
>>> data.day
1
>>> data.month
11
>>> data.year
2018
```

O tipo `datetime`, além das propriedades da data (`date`), permite acessar a hora, os minutos e os segundos independentemente:

```
>>> momento = datetime.datetime.now()
>>> momento.date()
datetime.date(2018, 11, 1)
>>> momento.time()
datetime.time(13, 46, 17, 94101)
>>> momento.hour
13
>>> momento.minute
46
>>> momento.second
17
```

e mesmo microssegundos:

```
>>> momento.microsecond
94101
```

Ou o dia da semana:

```
>>> momento.weekday()
3
```

No caso, 3 representa a quinta-feira, pois começamos a contar de 0 e a partir de segunda-feira. Para obter um valor mais compatível com o que usamos no Brasil, utilize `isoweekday`:

```
>>> momento.isoweekday()
4
```

Em que a quinta-feira é representada como 4, pois, além de começarmos a contar do 0, zero é domingo.

Uma conversão muito usada na web e principalmente para converter data e hora em string é o formato ISO 8601:

```
>>> momento.isoformat()
'2018-11-01T13:46:17.094101'
```

Que representa uma data no formato ano, mês, dia, hora, minutos, segundos e microssegundos.

Novos objetos podem ser criados passando valores como em:

```
>>> data = datetime.date(year=2019, month=9, day=7)
>>> data
datetime.date(2019, 9, 7)
```

Podemos também calcular valores futuros ou passados usando `timedelta`:

```
>>> daqui_a_90_dias = momento + datetime.timedelta(days=90)
>>> daqui_a_90_dias
datetime.datetime(2019, 1, 30, 13, 46, 17, 94101)
```

e mesmo realizar operações como:

```
>>> momento - datetime.timedelta(minutes=30)
datetime.datetime(2018, 11, 1, 13, 16, 17, 94101)
```

Para evitar repetir o nome datetime, você também pode importar apenas o tipo que vai usar:

```
>>> from datetime import datetime
>>> datetime.now()
datetime.datetime(2018, 11, 1, 14, 5, 16, 573056)
```

Python possui inúmeras funções e métodos para trabalhar com datas, horas e mesmo fusos horários. Consulte a documentação padrão para ter mais detalhes.

# CAPÍTULO 10

# Classes e objetos

A programação orientada a objetos facilita a escrita e a manutenção de nossos programas, utilizando classes e objetos. Classes são a definição de um novo tipo de dados que associa dados e operações em uma só estrutura. Um objeto pode ser entendido como uma variável cujo tipo é uma classe, ou seja, um objeto é uma instância de uma classe.

A programação orientada a objetos é uma técnica de programação que organiza nossos programas em classes e objetos em vez de apenas funções, como vimos até agora. É um assunto muito importante e extenso, merecendo vários livros e muita prática para ser completamente entendido. O objetivo deste capítulo é apresentar o básico da orientação a objetos de forma a introduzir o conceito e estimular o aprendizado dessa técnica.

## 10.1 Objetos como representação do mundo real

Podemos entender um objeto em Python como a representação de um objeto do mundo real, escrita em uma linguagem de programação. Essa representação é limitada pela quantidade de detalhes que podemos ou queremos representar, uma abstração.

Vejamos, por exemplo, um aparelho de televisão. Podemos dizer que uma televisão tem uma marca e um tamanho de tela. Podemos também pensar no que podemos fazer com esse aparelho, por exemplo mudar de canal, ligá-lo ou desligá-lo. Vejamos como escrever isso em Python:

```
>>> class Televisão: ❶
    def __init__(self): ❷
        self.ligada = False ❸
        self.canal = 2 ❹
```

```
>>> tv = Televisão() ❺
>>> tv.ligada ❻
False
>>> tv.canal
2
>>> tv_sala = Televisão() ❼
>>> tv_sala.ligada = True ❽
>>> tv_sala.canal = 4 ❾
>>> tv.canal
2
>>> tv_sala.canal
4
```

Em ❶, criamos uma nova classe chamada `Televisão`. Utilizamos a instrução **`class`** para indicar a declaração de uma nova classe e : para iniciar seu bloco. Quando declaramos uma classe, estamos criando um novo tipo de dados. Esse novo tipo define seus próprios métodos e atributos. Lembre-se dos tipos string e **`list`**. Esses dois tipos predefinidos do Python são classes. Quando criamos uma lista ou uma string, estamos instanciando ou criando uma instância dessas classes, ou seja, um objeto. Quando definimos nossas próprias classes, podemos criar nossos próprios métodos e atributos.

Em ❷, definimos um método especial chamado `__init__`. Métodos nada mais são que funções associadas a uma classe. O método `__init__` será chamado sempre que criarmos objetos da classe `Televisão`, sendo por isso chamado de construtor (*constructor*) ou inicializador. Um método construtor é chamado sempre que um objeto da classe é instanciado. É o construtor que inicializa nosso novo objeto com seus valores-padrão. O método `__init__` recebe um parâmetro chamado **`self`**. Por enquanto, entenda **`self`** como o objeto televisão em si, o que ficará mais claro adiante.

Em ❸, dizemos que **`self.ligada`** é um valor de **`self`**, ou seja, do objeto televisão. Todo método em Python tem **`self`** como primeiro parâmetro. Dizemos que **`self.ligada`** é um atributo do objeto. Como **`self`** representa o objeto em si, escreveremos **`self.ligada`**. Sempre que quisermos especificar atributos de objetos, devemos associá-los a **`self`**. Caso contrário, se escrevêssemos `ligada = False`, ligada seria apenas uma variável local do método `__init__`, e não um atributo do objeto.

Em ❹, dizemos que `canal` também é um valor ou característica de nossa televisão. Observe também que escrevemos **`self.canal`** para criar um atributo, e não uma simples variável local.

Em ❺ criamos um objeto tv utilizando a classe Televisão. Dizemos que tv é agora um objeto da classe Televisão ou que tv é uma instância de Televisão. Quando solicitamos a criação de um objeto, o método construtor de sua classe é chamado, em Python, __init__, como declaramos em ❷. Em ❻, exibimos o valor do atributo ligada e canal do objeto tv.

Já em ❼ criamos outra instância da classe Televisão chamada tv_sala. Em ❽, mudamos o valor de ligada para True e o canal para 4 em ❾.

Observe que ao imprimirmos o canal de cada televisão temos valores independentes, pois tv e tv_sala são dois objetos independentes, podendo cada um ter seus próprios valores, como duas televisões no mundo real. Quando criamos um objeto de uma classe, ele tem todos os atributos e métodos que especificamos ao declarar a classe e que foram inicializados em seu construtor. Essa característica simplifica o desenvolvimento dos programas, pois podemos definir o comportamento de todos os objetos de uma classe (métodos), preservando os valores individuais de cada um (atributos).

**Exercício 10.1** Adicione os atributos tamanho e marca à classe Televisão. Crie dois objetos Televisão e atribua tamanhos e marcas diferentes. Depois, imprima o valor desses atributos de forma a confirmar a independência dos valores de cada instância (objeto).

### Trivia

Python é uma linguagem que constrói seus objetos em duas etapas. A primeira etapa é realizada pelo método __new__, que em seguida chama o método __init__. Essa construção em duas etapas não é comum em outras linguagens de programação e gera problemas ao se identificar corretamente qual dos dois métodos é o construtor da classe (*). No caso de Python, __new__ cria a instância e é utilizado para alterar a definição de classe dinamicamente, em geral usada em programas mais avançados e frameworks. __init__ é chamado por __new__ para inicializar a classe com seus valores iniciais. Isso ocasiona o problema de chamar __init__ de construtor. Como não utilizaremos __new__ neste livro e para manter a equivalência com outras linguagens de programação, chamaremos __init__ de construtor ou de inicializador.

(*) https://docs.python.org/3/reference/datamodel.html

Vejamos agora como associar um comportamento à classe Televisão, definindo dois métodos muda_canal_para_cima e muda_canal_para_baixo:

```
>>> class Televisão:
...     def __init__(self):
...         self.ligada = False
...         self.canal = 2
...     def muda_canal_para_baixo(self): ❶
...         self.canal -= 1
...     def muda_canal_para_cima(self): ❷
...         self.canal += 1
...
>>> tv = Televisão()
>>> tv.muda_canal_para_cima() ❸
>>> tv.muda_canal_para_cima()
>>> tv.canal
4
>>> tv.muda_canal_para_baixo() ❹
>>> tv.canal
3
```

Em ❶, definimos o método `muda_canal_para_baixo`. Observe que não utilizamos _ antes do nome do método, pois esse nome não é um nome especial do Python, mas apenas um nome escolhido por nós. Veja que passamos também um parâmetro **self**, que representa o objeto em si. Observe que escrevemos diretamente `self.canal -= 1`, utilizando o atributo `canal` da televisão. Isso é possível porque criamos o atributo `canal` no construtor(`__init__`). É usando atributos que podemos armazenar valores entre as chamadas dos métodos.

Em ❷, fizemos a mesma coisa, mas, dessa vez, com `muda_canal_para_cima`.

Em ❸, chamamos o método. Observe que escrevemos o nome do método após o nome do objeto, separando-os com um ponto, bem como que o método foi chamado da mesma forma que uma função, mas que na chamada não passamos nenhum parâmetro. Na realidade, o interpretador Python adiciona o objeto `tv` à chamada, utilizando-o como o **self** do método em ❷. É assim que o interpretador consegue trabalhar com vários objetos de uma mesma classe.

Depois, em ❹, fazemos a chamada do método `muda_canal_para_baixo`. Veja o valor retornado por `tv.canal`, antes e depois de chamarmos o método.

A grande vantagem de usar classes e objetos é facilitar a construção dos programas. Embora simples, você pode observar que não precisamos enviar o canal atual da televisão ao método `muda_canal_para_cima`, simplificando a chamada do método. Esse efeito "memória" facilita a configuração de objetos complexos, pois

armazenamos as características importantes em seus atributos, evitando repassar esses valores a cada chamada.

Na verdade, esse tipo de construção imita o comportamento do objeto no mundo real. Quando mudamos o canal da tv para cima ou para baixo, não informamos o canal atual para televisão!

## 10.2 Passagem de parâmetros

Um problema com a classe televisão é que não controlamos os limites de nossos canais. Na realidade, podemos até obter canais negativos ou números muito grandes, como 35790. Vamos modificar o construtor de forma a receber o canal mínimo e máximo suportado por tv:

```
class Televisão:
    def __init__(self, min, max):
        self.ligada = False
        self.canal = 2
        self.cmin = min
        self.cmax = max
    def muda_canal_para_baixo(self):
        if self.canal - 1 >= self.cmin:
            self.canal -= 1
    def muda_canal_para_cima(self):
        if self.canal + 1 <= self.cmax:
            self.canal += 1
tv = Televisão(1, 99)
for x in range(0, 120):
    tv.muda_canal_para_cima()
print(tv.canal)
for x in range(0, 120):
    tv.muda_canal_para_baixo()
print(tv.canal)
```

Execute o programa e verifique se as modificações deram resultado. Observe que mudamos o comportamento da classe `Televisão` sem mudar quase nada no programa que a utiliza. Isso porque isolamos os detalhes do funcionamento da classe do resto do programa. Esse efeito é chamado de encapsulamento. Dizemos que uma classe deve encapsular ou ocultar detalhes de seu funcionamento o máximo possível. No caso dos métodos para mudar o canal, incluímos a verificação sem alterar o resto do programa. Simplesmente solicitamos que o método

realize seu trabalho, sem se preocupar com os detalhes internos de como ele realizará essa operação.

**Exercício 10.2** Atualmente, a classe `Televisão` inicializa o canal com 2. Modifique a classe Televisão de forma a receber o canal inicial em seu construtor.

**Exercício 10.3** Modifique a classe `Televisão` de forma que, se pedirmos para mudar o canal para baixo, além do mínimo, ela vá para o canal máximo. Se mudarmos para cima, além do canal máximo, que volte ao canal mínimo. Exemplo:

```
>>> tv = Televisão(2, 10)
>>> tv.muda_canal_para_baixo()
>>> tv.canal
10
>>> tv.muda_canal_para_cima()
>>> tv.canal
2
```

Ao trabalharmos com classes e objetos, assim como fizemos ao estudar funções, precisamos representar em Python uma abstração do problema. Quando realizamos uma abstração, reduzimos os detalhes do problema ao necessário para solucioná-lo. Estamos construindo um modelo, ou seja, modelando nossas classes e objetos.

Antes de continuarmos, você deve entender que o modelo pode variar de uma pessoa para outra, como todas as partes do programa. Um dos detalhes mais difíceis é decidir o quanto representar e onde limitar os modelos. No exemplo da classe `Televisão` não escrevemos nada sobre a tomada da TV, se esta tem controle remoto, em que parte da casa está localizada ou mesmo se tem controle de volume.

Como regra simples, modele apenas as informações que você precisa, adicionando detalhes à medida que for necessário. Com o tempo, a experiência ensinará quando parar de detalhar seu modelo, como também apontará erros comuns.

Tudo que aprendemos com funções é também válido para métodos. A principal diferença é que um método é associado a uma classe e atua sobre um objeto. O primeiro parâmetro do método é chamado **`self`** e representa a instância sobre a qual o método atuará. É por meio de **`self`** que teremos acesso aos outros métodos de uma classe, preservando todos os atributos de nossos objetos. Você não precisa passar o objeto como primeiro parâmetro ao invocar (chamar) um método: o interpretador Python faz isso automaticamente para você. Entretanto, não se esqueça de declarar **`self`** como o primeiro parâmetro de seus métodos.

**Exercício 10.4** Utilizando o que aprendemos com funções, modifique o construtor da classe `Televisão` de forma que `min` e `max` sejam parâmetros opcionais, em que `min` vale 2 e `max` vale 14, caso outro valor não seja passado.

**Exercício 10.5** Utilizando a classe `Televisão` modificada no exercício anterior, crie duas instâncias (objetos), especificando o valor de `min` e `max` por nome.

## 10.3 Exemplo de um banco

Vejamos o exemplo de classes, mas, dessa vez, vamos modelar contas correntes de um banco. Imagine o Banco Tatu, moderno e eficiente, mas precisando de um novo programa para controlar o saldo de seus correntistas. Cada conta corrente pode ter um ou mais clientes como titular. O banco controla apenas o nome e o telefone de cada cliente. A conta corrente apresenta um saldo e uma lista de operações de saques e depósitos. Quando o cliente fizer um saque, diminuiremos o saldo da conta corrente. Quando ele fizer um depósito, aumentaremos o saldo. Por enquanto, o Banco Tatu não oferece contas especiais, ou seja, o cliente não pode sacar mais dinheiro que seu saldo permite.

Vamos resolver esse problema por partes. A classe `Cliente` é simples, tendo apenas dois atributos: nome e telefone. Digite o programa a seguir e salve-o em um arquivo chamado `cliente.py`.

Classe `Clientes` (`clientes.py`):

```
class Cliente:
    def __init__(self, nome, telefone):
        self.nome = nome
        self.telefone = telefone
```

Abra o arquivo `clientes.py` no IDLE e execute-o (Run – F5). No interpretador, experimente criar um objeto da classe `Cliente`.

```
joão = Cliente("João da Silva", "777-1234")
maria = Cliente("Maria Silva", "555-4321")
joão.nome
joão.telefone
maria.nome
maria.telefone
```

Como nos outros programas Python, podemos executar uma definição de classe dentro de um arquivo `.py` e utilizar o interpretador para experimentar nossas

classes e objetos. Essa operação é muito importante para testarmos as classes, modificar alguns valores e repetir o teste.

Como o programa do Banco Tatu vai ficar maior que os programas que já trabalhamos até aqui, vamos gravar cada classe em um arquivo `.py` separado. Em Python essa organização não é obrigatória, pois a linguagem permite que tenhamos todas as nossas classes em um só arquivo, se assim quisermos.

Vamos agora criar o arquivo `teste.py`, que vai simplesmente importar `clientes.py` e criar dois objetos:

```
from clientes import Cliente
joão = Cliente("João da Silva", "777-1234")
maria = Cliente("Maria da Silva", "555-4321")
```

Observe que com poucas linhas de código conseguimos reutilizar a classe `Cliente`. Isso é possível porque `clientes.py` e `teste.py` estão no mesmo diretório. Essa separação permite que passemos a experimentar as novas classes no interpretador, armazenando o **`import`** e a criação dos objetos em um arquivo à parte. Assim poderemos definir nossas classes separadamente dos experimentos ou testes.

Para resolver o problema do Banco Tatu, precisamos de outra classe, `Conta`, para representar uma conta do banco com seus clientes e seu saldo; salve-o com o nome `contas.py`:

```
class Conta:
    def __init__(self, clientes, número, saldo=0):
        self.saldo = saldo
        self.clientes = clientes
        self.número = número
    def resumo(self):
        print(f"CC Número: {self.número} Saldo: {self.saldo:10.2f}")
    def saque(self, valor):
        if self.saldo >= valor:
            self.saldo -= valor
    def deposito(self, valor):
        self.saldo += valor
```

A classe `Conta` é definida recebendo clientes, número e saldo em seu construtor (`__init__`), onde em clientes esperamos uma lista de objetos da classe `Cliente`. `número` é uma string com o número da conta, e saldo é um parâmetro opcional, tendo zero (0) como padrão. A listagem também apresenta os métodos `resumo`, `saque` e `depósito`. O método `resumo` exibe na tela o número da conta corrente e seu saldo. `saque` permite retirar dinheiro da conta corrente, verificando se essa

operação é possível (`self.saldo >= valor`). `depósito` simplesmente adiciona o valor solicitado ao saldo da conta corrente.

Altere o programa `teste.py` de forma a importar a classe `Conta`. Crie uma conta corrente para os clientes João e Maria.

Faça alguns testes no interpretador:

```
conta.resumo()
conta.saque(1000)
conta.resumo()
conta.saque(50)
conta.resumo()
conta.depósito(200)
conta.resumo()
```

Embora nossa conta corrente comece a funcionar, ainda não temos a lista de operações de cada elemento. Essa lista é, na realidade, um extrato de conta. Vamos alterar a classe `Conta` de forma a adicionar um atributo que é a lista de operações realizadas. Considere o saldo inicial como um depósito. Vamos adicionar também um método extrato para imprimir todas as operações realizadas:

```
# Programa 10.1 - Conta com registro de operações e extrato (contas.py)
class Conta:
    def __init__(self, clientes, número, saldo=0):
        self.saldo = 0
        self.clientes = clientes
        self.número = número
        self.operações = []
        self.deposito(saldo)
    def resumo(self):
        print(f"CC N°{self.número} Saldo: {self.saldo:10.2f}")
    def saque(self, valor):
        if self.saldo >= valor:
            self.saldo -= valor
            self.operações.append(["SAQUE", valor])
    def deposito(self, valor):
        self.saldo += valor
        self.operações.append(["DEPÓSITO", valor])
    def extrato(self):
        print(f"Extrato CC N° {self.número}\n")
        for o in self.operações:
            print(f"{o[0]:10s} {o[1]:10.2f}")
        print(f"\n    Saldo: {self.saldo:10.2f}\n")
```

Modifique também o programa de testes para imprimir o extrato de cada conta:

```
from clientes import Cliente
from contas import Conta
joão = Cliente("João da Silva", "777-1234")
maria = Cliente("Maria da Silva", "555-4321")
conta1 = Conta([joão], 1, 1000)
conta2 = Conta([maria, joão], 2, 500)
conta1.saque(50)
conta2.deposito(300)
conta1.saque(190)
conta2.deposito(95.15)
conta2.saque(250)
conta1.extrato()
conta2.extrato()
```

**Exercício 10.6** Altere o programa de forma que a mensagem saldo insuficiente seja exibida caso haja tentativa de sacar mais dinheiro que o saldo disponível.

**Exercício 10.7** Modifique o método resumo da classe `Conta` para exibir o nome e o telefone de cada cliente.

**Exercício 10.8** Crie uma nova conta, agora tendo João e José como clientes e saldo igual a 500.

Para resolver o problema do Banco Tatu, precisamos de uma classe para armazenar todas as nossas contas. Como atributos do banco, teríamos seu nome e a lista de contas. Como operações, considere a abertura de uma conta corrente e a listagem de todas as contas do banco. A classe `Banco`:

```
class Banco:
    def __init__(self, nome):
        self.nome = nome
        self.clientes = []
        self.contas = []
    def abre_conta(self, conta):
        self.contas.append(conta)
    def lista_contas(self):
        for c in self.contas:
            c.resumo()
```

Agora, vamos criar os objetos:

```
from clientes import Cliente
from bancos import Banco
from contas import Conta
joão = Cliente("João da Silva", "3241-5599")
maria = Cliente("Maria Silva", "7231-9955")
josé = Cliente("José Vargas", "9721-3040")
contaJM = Conta([joão, maria], 100)
contaJ = Conta([josé], 10)
tatu = Banco("Tatú")
tatu.abre_conta(contaJM)
tatu.abre_conta(contaJ)
tatu.lista_contas()
```

**Exercício 10.9** Crie classes para representar estados e cidades. Cada estado tem um nome, sigla e cidades. Cada cidade tem nome e população. Escreva um programa de testes que crie três estados com algumas cidades em cada um. Exiba a população de cada estado como a soma da população de suas cidades.

## 10.4 Herança

A orientação a objetos permite modificar nossas classes, adicionando ou modificando atributos e métodos, tendo como base outra classe. Vejamos o exemplo do Banco Tatu, em que o novo sistema foi um sucesso. Para atrair novos clientes, o Banco Tatu começou a oferecer contas especiais aos clientes. Uma conta especial permite que possamos sacar mais dinheiro que atualmente disponível no saldo da conta, até um determinado limite. As operações de depósito, extrato e resumo são as mesmas de uma conta normal. Vamos criar a classe `ContaEspecial` herdando o comportamento da classe `Conta`. Digite o programa a seguir no mesmo arquivo em que a classe `Conta` foi definida (`contas.py`):

```
class ContaEspecial(Conta):❶
    def __init__(self, clientes, número, saldo=0, limite=0):
        Conta.__init__(self, clientes, número, saldo) ❷
        self.limite = limite ❸
    def saque(self, valor):
        if self.saldo + self.limite >= valor:
            self.saldo -= valor
            self.operações.append(["SAQUE", valor])
```

Em ❶ definimos a classe `ContaEspecial`, mas observe que escrevemos `Conta` entre parênteses. Esse é o formato de declaração de classe usando herança, ou seja, é assim que declaramos a herança de uma classe em Python. Essa linha diz: crie uma nova classe chamada `ContaEspecial` herdando todos os métodos e atributos da classe `Conta`. A partir daqui, `ContaEspecial` é uma subclasse de `Conta`. Também dizemos que `Conta` é a superclasse de `ContaEspecial`.

Em ❷, chamamos o método `__init__` de `Conta`, escrevendo `Conta.__init__` seguido dos parâmetros que normalmente passaríamos. Toda vez que você utilizar herança, o método construtor da superclasse, no caso `Conta`, deve ser chamado. Observe que, nesse caso, passamos **`self`** para `Conta.__init__`. É assim que reutilizamos as definições já realizadas na superclasse, evitando ter de reescrever as atribuições de clientes, número e saldo. Chamar a inicialização da superclasse também tem outras vantagens, como garantir que modificações no construtor da superclasse não tenham de ser duplicadas em todas as subclasses.

Em ❸ criamos o atributo **`self`**`.limite`. Esse atributo será criado apenas para classes do tipo `ContaEspecial`. Observe que criamos o novo atributo depois de chamarmos `Conta.__init__`.

Observe também que não chamamos `Conta.saque` no método `saque` de `ContaEspecial`. Quando isso ocorre, estamos substituindo completamente a implementação do método por uma nova. Uma das grandes vantagens de utilizar herança de classes é justamente poder substituir ou complementar métodos já definidos.

Modifique seu programa de teste para que se pareça com o programa:

```
from clientes import Cliente
from contas import Conta, ContaEspecial ❶
joão = Cliente("João da Silva", "777-1234")
maria = Cliente("Maria da Silva", "555-4321")
conta1 = Conta([joão], 1, 1000)
conta2 = ContaEspecial([maria, joão], 2, 500, 1000) ❷
conta1.saque(50)
conta2.deposito(300)
conta1.saque(190)
conta2.deposito(95.15)
conta2.saque(1500)
conta1.extrato()
conta2.extrato()
```

Vejamos o que mudou no programa de testes. Em ❶ adicionamos o nome da classe `ContaEspecial` ao **`import`**. Dessa forma, poderemos utilizar a nova classe em

nossos testes. Já em ❷ criamos um objeto `ContaEspecial`. Veja que, praticamente, não mudamos nada, exceto o nome da classe, que agora é `ContaEspecial`, e não `Conta`. Um detalhe importante para o teste é que adicionamos um parâmetro ao construtor, no caso 1000, como o valor de `limite`. Execute esse programa de teste e observe que, para a `conta2`, obtivemos um saldo negativo.

Utilizando herança, modificamos muito pouco nosso programa, mantendo a funcionalidade anterior e adicionando novos recursos. O interessante de tudo isso é que foi possível reutilizar os métodos que já havíamos definido na classe `Conta`. Isso permitiu que a definição da classe `ContaEspecial` fosse bem menor, pois lá especificamos apenas o comportamento que é diferente.

Quando você utilizar herança, tente criar classes nas quais o comportamento e as características comuns fiquem na superclasse. Dessa forma, você poderá definir subclasses enxutas. Outra vantagem de utilizar herança é que, se mudarmos algo na superclasse, essas mudanças serão também usadas pelas subclasses. Um exemplo seria modificarmos o método de extrato. Como em `ContaEspecial` não especificamos um novo método de extrato, ao modificarmos o método `Conta.extrato`, estaremos também modificando o extrato de `ContaEspecial`, pois as duas classes compartilham o mesmo método.

É importante notar que, ao utilizarmos herança, as subclasses devem poder substituir suas superclasses, sem perda de funcionalidade e sem gerar erros nos programas. O importante é que você conheça esse novo recurso e comece a utilizá-lo em seus programas. Lembre-se de que você não é obrigado a definir uma hierarquia de classes em todos os seus programas. Com o tempo, a necessidade de utilizar herança ficará mais clara.

**Exercício 10.10** Modifique as classes `Conta` e `ContaEspecial` para que a operação de saque retorne verdadeiro se o saque foi efetuado e falso, caso contrário.

**Exercício 10.11** Altere a classe `ContaEspecial` de forma que seu extrato exiba o limite e o total disponível para saque.

**Exercício 10.12** Observe o método saque das classes `Conta` e `ContaEspecial`. Modifique o método `saque` da classe `Conta` de forma que a verificação da possibilidade de saque seja feita por um novo método, substituindo a condição atual. Esse novo método retornará verdadeiro se o saque puder ser efetuado, e falso, caso contrário. Modifique a classe `ContaEspecial` de forma a trabalhar com esse novo método. Verifique se você ainda precisa trocar o método saque de `ContaEspecial` ou apenas o novo método criado para verificar a possibilidade de saque.

## 10.5 Desenvolvendo uma classe para controlar listas

Agora que já temos uma ideia de como utilizar classes em Python, vamos criar uma classe que controle uma lista de objetos. Nos exemplos anteriores, ao usarmos a lista de clientes como uma lista simples, não fizemos nenhuma verificação quanto à duplicidade de valores. Por exemplo, uma lista de clientes em que os dois clientes são a mesma pessoa seria aceita sem problemas. Outro problema de nossa lista simples é que ela não verifica se os elementos são objetos da classe `Cliente`. Vamos modificar isso construindo uma nova classe, chamada `ListaÚnica`:

```
class ListaÚnica:
    def __init__(self, elem_class):
        self.lista = []
        self.elem_class = elem_class
    def __len__(self):
        return len(self.lista)
    def __iter__(self):
        return iter(self.lista)
    def __getitem__(self, p):
        return self.lista[p]
    def indiceVálido(self, i):
        return i >= 0 and i < len(self.lista)
    def adiciona(self, elem):
        if self.pesquisa(elem) == -1:
            self.lista.append(elem)
    def remove(self, elem):
        self.lista.remove(elem)
    def pesquisa(self, elem):
        self.verifica_tipo(elem)
        try:
            return self.lista.index(elem)
        except ValueError:
            return -1
    def verifica_tipo(self, elem):
        if not isinstance(type(elem), self.elem_class):
            raise TypeError("Tipo inválido")
    def ordena(self, chave=None):
        self.lista.sort(key=chave)
```

Vejamos o que podemos fazer com essa classe, realizando alguns testes no interpretador:

```
>>> from listaunica import *
>>> lu = ListaÚnica(int) ❶
>>> lu.adiciona(5) ❷
>>> lu.adiciona(3)
>>> lu.adiciona(2.5) ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "listaunica.py", line 19, in adiciona
    if self.pesquisa(elem) == -1:
  File "listaunica.py", line 26, in pesquisa
    self.verifica_tipo(elem)
  File " listaunica.py", line 34, in verifica_tipo
    raise TypeError("Tipo inválido")
TypeError: Tipo inválido
```

Em ❶ criamos um objeto com a classe `ListaÚnica` que acabamos de importar. Veja que passamos **`int`** como parâmetro no construtor, indicando que essa lista deve apenas conter valores do tipo **`int`** (inteiros). Em ❷, adicionamos o número inteiro 5 como elemento de nossa lista, assim como 3 na linha seguinte. Veja que não tivemos algum problema, mas que em ❸, ao tentarmos adicionar 2.5 (um número do tipo **`float`**), obtivemos uma exceção com mensagem de erro tipo inválido (`TypeError`). Isso acontece porque, em nosso construtor, o parâmetro `elem` é na realidade a classe que desejamos para nossos elementos. Sempre que um elemento é adicionado (método `adiciona`), uma verificação do tipo do novo elemento é feita pelo método `verifica_tipo`, que gera uma exceção, caso o tipo não seja igual ao tipo passado ao construtor. Veja que o método `adiciona` também realiza uma pesquisa para verificar se um elemento igual já não faz parte da lista e só realiza a adição caso um elemento igual não tenha sido encontrado. Dessa forma, podemos garantir que nossa lista conterá apenas elementos do mesmo tipo e sem repetições.

Vamos continuar a testar a nossa nova classe no interpretador:

```
>>> len(lu) ❹
2
>>> for e in lu: ❺
...     print(e)
...
5
3
>>> lu.adiciona(5) ❻
>>> len(lu) ❼
```

```
2
>>> lu[0] ❽
5
>>> lu[1]
3
```

Em ❹, utilizamos a função **len** com nosso objeto e obtivemos o resultado esperado. Veja que não tivemos de escrever **len**(lu.lista), mas apenas **len**(lu). Isso é possível, pois implementamos o método __len__, que é responsável por retornar o número de elementos a partir de **self**.lista.

Em ❺, utilizamos nosso objeto lu em um **for**. Isso foi possível com a implementação do método __iter__, que é chamado quando utilizamos um objeto com **for**. O método __iter__ simplesmente chama a função **iter** do Python com nossa lista interna **self**.lista. A intenção de utilizarmos esses métodos é de esconder alguns detalhes de nossa classe ListaÚnica e evitar o acesso direto à **self**.lista.

Em ❻, testamos a inclusão de um elemento repetido para, na linha seguinte, em ❼, confirmarmos que a quantidade de elementos da lista não foi alterada. Isso acontece porque, ao realizar a pesquisa no método adiciona, o valor é encontrado, e a adição é ignorada.

Em ❽, utilizamos índices com nossa classe, da mesma forma que fazemos com uma lista normal do Python. Isso é possível, pois implementamos o método __getitem__, que recebe o valor do índice (p) e retorna o elemento correspondente de nossa lista.

Python possui vários métodos mágicos, métodos especiais que têm o formato __nome__. O método __init__, usado em nossos construtores, é um método mágico, __len__, __getitem__ e __iter__ também. Esses métodos permitem dar outro comportamento a nossas classes e usá-las quase que como classes da própria linguagem. A utilização desses métodos mágicos não é obrigatória, mas possibilita uma grande flexibilidade para nossas classes.

Vejamos outro exemplo de classe com métodos mágicos (especiais). A classe Nome é utilizada para guardar nomes de pessoas e manter uma chave de pesquisa. Grave o programa a seguir como nome.py:

```
class Nome:
    def __init__(self, nome):
        if nome is None or not nome.strip():
            raise ValueError("Nome não pode ser nulo nem em branco")
        self.nome = nome
        self.chave = nome.strip().lower()
```

```
    def __str__(self):
        return self.nome
    def __repr__(self):
        return f"<Classe {type(self).__name__} em 0x{id(self):x} Nome: {self.nome}
    Chave: {self.chave }>"
    def __eq__(self, outro):
        print("__eq__ Chamado")
        return self.nome == outro.nome
    def __lt__(self, outro):
        print("__lt__ Chamado")
        return self.nome < outro.nome
```

Na classe `Nome`, definimos o método `__init__` de forma a gerar uma exceção caso a string passada como nome seja nula (`None`) ou em branco. Veja que implementamos também o método mágico `__str__`, que é chamado ao imprimirmos um objeto da classe `Nome`. Dessa forma, podemos configurar a saída de nossas classes com mais liberdade e sem mudar o comportamento esperado de uma classe normal em Python.

Vejamos o resultado desse programa no interpretador:

```
>>> from nome import Nome
>>> A = Nome("Nilo") ❶
>>> print(A) ❷
Nilo
>>> B = Nome("  ") ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "nome.py", line 4, in __init__
    raise ValueError("Nome não pode ser nulo nem em branco")
ValueError: Nome não pode ser nulo nem em branco
>>> C = Nome(None) ❹
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "nome.py", line 4, in __init__
    raise ValueError("Nome não pode ser nulo nem em branco")
ValueError: Nome não pode ser nulo nem em branco
```

Em ❶, criamos um objeto da classe `Nome`, passando a string `"Nilo"`. Em ❷, imprimimos o objeto `A` diretamente, veja que o resultado foi o nome Nilo, que é o valor retornado pelo nosso método `__str__`. Em ❸ e ❹, testamos a inicialização de um objeto com valores inválidos. Veja que uma exceção do tipo `ValueError` foi

gerada, impedindo a criação do objeto com o valor inválido. O objetivo desse tipo de validação é garantir que o valor usado na classe seja válido. Podemos implementar regras mais complexas dependendo do objetivo de nosso programa, mas, para esse exemplo, verificar se a string está em branco ou se é nula (**None**) é suficiente.

Vamos continuar a descobrir os outros métodos especiais no interpretador:

```
>>> A = Nome("Nilo")
>>> A ❶
<Classe Nome em 0x26ee8f0 Nome: Nilo Chave: nilo>
>>> A == Nome("Nilo") ❷
__eq__ Chamado
True
>>> A != Nome("Nilo") ❸
__eq__ Chamado
False
>>> A < Nome("Nilo") ❹
__lt__ Chamado
False
>>> A > Nome("Nilo") ❺
__lt__ Chamado
False
```

Em ❶, temos a representação do objeto `A`, ou seja, a forma que é usada pelo interpretador para mostrar o objeto fora da função **`print`** ou quando usamos a função **`repr`**. Veja que a saída foi gerada pelo método `__repr__` da classe `Nome`.

Em ❷, utilizamos o operador == para verificar se o objeto `A` é igual a outro objeto. Em Python, o comportamento padrão é que == retorne **`True`** se os dois objetos forem o mesmo objeto, ou **`False`** caso contrário. No entanto, o objeto `A` e `Nome("Nilo")` são claramente duas instâncias diferentes da classe `Nome`. Veja que o resultado é **`True`** e que nosso método `__eq__` foi chamado. Isso acontece porque `__eq__`(*equal*) é o método especial utilizado para comparações de igualdade (==) de nossos objetos. Em nossa implementação, retornamos **`True`** se o conteúdo de **`self`**`.nome` for igual nos dois objetos, independentemente de serem o mesmo objeto ou não. Veja também que, em ❸, o método `__eq__` também foi chamado, embora não tenhamos definido o método `__neq__`(`!=`, *not equal*, não igual). Esse comportamento vem da implementação padrão desses métodos, em que `__eq__` foi negado para implementar o `__neq__` inexistente. Veremos depois uma forma de declarar todos os operadores de comparação apenas implementando o `__eq__` e `__lt__`(*less than*, menor que).

Em ❹, usamos o operador < (menor que) para comparar os dois objetos. Veja que o método especial `__lt__` foi chamado nesse caso. Nossa implementação do método `__lt__` utiliza a comparação de `self.nome` para decidir a ordem dos objetos. Em ❺, o operador > (maior que) retorna o valor correto, e também chama o método `__lt__`, uma vez que o método `__gt__` (*greater than*, maior que) não foi implementado, de forma análoga ao que aconteceu entre `__eq__` e `__neq__`.

Agora veja o que acontece quando tentamos utilizar os operadores >= (maior ou igual) e <= (menor ou igual):

```
>>> A >= Nome("Nilo")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Nome() >= Nome()
>>> A <= Nome("Nilo")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Nome() <= Nome()
```

Isso acontece porque os operadores >= e <= chamam os métodos `__ge__` (*greater than or equal*, maior ou igual) e `__le__` (*less than or equal*, menor ou igual), respectivamente, e ambos não foram implementados.

Como algumas dessas relações não funcionam como o esperado, existe uma forma mais simples de implementar esses métodos especiais, apenas com a implementação de `__eq__` e de `__lt__`. Vamos usar um recurso chamado *decorators* (adornos ou decoradores):

```
from functools import total_ordering

@total_ordering
class Nome:
    def __init__(self, nome):
        if nome is None or not nome.strip():
            raise ValueError("Nome não pode ser nulo nem em branco")
        self.nome = nome
        self.chave = Nome.CriaChave(nome)
    def __str__(self):
        return self.nome
    def __repr__(self):
        return f"<Classe {type(self).__name__} em 0x{id(self):x} Nome: {self.nome
    } Chave: { self.chave }>"
```

```
    def __eq__(self, outro):
        print("__eq__ Chamado")
        return self.nome == outro.nome
    def __lt__(self, outro):
        print("__lt__ Chamado")
        return self.nome < outro.nome
    @staticmethod
    def CriaChave(nome):
        return nome.strip().lower()
```

O primeiro decorador `@total_ordering` é definido no módulo `functools`, por isso tivemos de importá-lo no início de nosso programa. Ele é responsável por implementar, ou seja, gerar o código responsável pela implementação de todos os métodos de comparação especiais, a partir de `__eq__` e de `__lt__`. Dessa forma, `__neq__` será a negação de `__eq__`; `__gt__`, a negação de `__lt__`; `__le__`, a combinação de `__lt__` com `__eq__`; e `__ge__`, a combinação de `__gt__` com `__eq__`, implementado, assim, todos os operadores de comparação (==, !=, >, <, >=, <=).

Vejamos o resultado no interpretador:

```
>>> from nome import Nome
>>> A = Nome("Nilo")
>>> A == Nome("Nilo")
__eq__ Chamado
True
>>> A != Nome("Nilo")
__eq__ Chamado
False
>>> A > Nome("Nilo")
__lt__ Chamado
__eq__ Chamado
False
>>> A < Nome("Nilo")
__lt__ Chamado
False
>>> A <= Nome("Nilo")
__lt__ Chamado
__eq__ Chamado
True
>>> A >= Nome("Nilo")
__lt__ Chamado
True
```

Observe que utilizamos um outro decorador **@staticmethod** antes da definição do método CriaChave. Veja também que o método CriaChave não possui o parâmetro **self**. Esse decorador cria um método estático, isto é, um método que pode ser chamado apenas com o nome da classe, não necessitando de um objeto para ser chamado. Vejamos no interpretador:

```
>>> A.CriaChave("X")
'x'
>>> Nome.CriaChave("X")
'x'
```

Tanto a chamada de CriaChave com o objeto A quanto a chamada com o nome da classe em Nome.CriaChave funcionaram como esperado. Métodos estáticos são utilizados para implementar métodos que não acessam ou que não precisam acessar as propriedades de uma instância da classe. No caso do método CriaChave, apenas o parâmetro nome é necessário para seu funcionamento. O decorador **@staticmethod** é necessário para informar ao interpretador não passar o parâmetro **self** automaticamente, como faz nos métodos normais, não estáticos. A vantagem de definirmos métodos estáticos é agrupar certas funções em nossas classes. Dessa forma, fica claro que CriaChave funciona no contexto de nomes. Se fosse definido como uma função fora de uma classe, sua implementação poderia parecer mais genérica do que realmente é. Dessa forma, métodos estáticos ajudam a manter o programa coeso e a manter o contexto da implementação do método.

O programa seguinte introduziu um problema de consistência de dados. Veja que, ao construirmos o objeto em __init__, verificamos o valor de nome e também atualizamos a chave de pesquisa. O problema é que podemos alterar nome sem qualquer validação e deixar chave em um estado inválido. Vejamos isso no interpretador:

```
>>> A = Nome("Teste")
>>> A
<Classe Nome em 0x3000f10 Nome: Teste Chave: teste>
>>> A.nome = "Nilo"
>>> A
<Classe Nome em 0x3000f10 Nome: Nilo Chave: teste>
>>> A.chave = "TST"
>>> A
<Classe Nome em 0x3000f10 Nome: Nilo Chave: TST>
```

Veja que, depois da construção do objeto, o nome e a chave estão corretos, pois essa operação é garantida pela nossa implementação de __init__. No entanto, um acesso à A.nome não atualizará o valor da chave, que continua com o valor configurado pelo construtor. O mesmo acontece com o atributo chave que pode

ser alterado sem qualquer ligação com o nome em si. Na programação orientada a objetos, uma classe é responsável por gerenciar seu estado interno e manter a consistência do objeto entre chamadas de métodos. Para garantir que nome e chave sempre sejam atualizados corretamente, vamos fazer uso de um recurso do Python que utiliza métodos para realizar a atribuição e a leitura de valores. Esse recurso é chamado de propriedade.

O programa seguinte utilizará um outro recurso, uma convenção especial de nomenclatura de nossos métodos para esconder os atributos da classe e impedir o acesso direto a eles de fora da classe:

```
from functools import total_ordering
@total_ordering
class Nome:
    def __init__(self, nome):
        self.nome = nome ❶
    def __str__(self):
        return self.nome
    def __repr__(self):
        return f"<Classe {type(self).__name__} em 0x{id(self):x} Nome: {self.__
    nome} Chave: {self.__chave}>" ❷
    def __eq__(self, outro):
        return self.nome == outro.nome
    def __lt__(self, outro):
        return self.nome < outro.nome
    @property ❸
    def nome(self):
        return self.__nome ❹
    @nome.setter ❺
    def nome(self, valor):
        if valor is None or not valor.strip():
            raise ValueError("Nome não pode ser nulo nem em branco")
        self.__nome = valor ❻
        self.__chave = Nome.CriaChave(valor) ❼
    @staticmethod
    def CriaChave(nome):
        return nome.strip().lower()
```

O programa utiliza outros decoradores como `@property`, em ❸, e `@nome.setter`, em ❺. Esses decoradores modificam o método logo abaixo deles, transformando-os em propriedades. O primeiro decorador, `@property`, transforma o método nome na

propriedade nome; dessa forma, sempre que escrevermos objeto.nome, estaremos chamando esse método, que retorna **self.**__nome. Já o segundo decorador, @nome.setter, transforma o método nome na propriedade usada para alterar o valor de __nome. Dessa forma, quando escrevermos objeto.nome = valor, esse método será chamado para efetuar as alterações. Observe que copiamos a verificação de tipo e a criação da chave do método __init__ (em ❶) para o método marcado por @nome.setter em ❺. Assim, em ❶, chamamos o método de ❺ ao escrevermos: **self.**nome = nome, uma vez que **self.**nome é agora uma propriedade. Outro detalhe importante é que acrescentamos duas sublinhas (_) antes do nome dos atributos nome e chave, fazendo-os __nome e __chave. Dessa forma, __nome e __chave ficam escondidos quando acessados de fora da classe. Esse "esconder" é apenas um detalhe da implementação do Python, que modifica o nome desses atributos de forma a torná-los inacessíveis (*name mangling*). No entanto, seus nomes foram apenas transformados pela adição de _ e do nome da classe, fazendo com que __chave se torne _Nome__chave. Mas essa proteção é suficiente para garantir que nome e chave estejam sincronizadas e que nossa classe funcione como esperado. Em Python, preste muita atenção ao utilizar nomes que começam com _ ou __. Esses símbolos indicam que esses atributos não devem ser acessados, exceto pelo código da própria classe. Não confundir os atributos protegidos, cujo nome começa por __, com o nome dos métodos mágicos os especiais que já vimos, que começam e terminam por __. Em ❷, podemos ver que __nome e __chave continuam acessíveis dentro do código da classe por **self.**__nome e **self.**__chave.

Vamos experimentar o novo código no interpretador:

```
>>> from nome import Nome
>>> A = Nome("Nilo") ❶
>>> A
<Classe Nome em 0x3140f10 Nome: Nilo Chave: nilo>
>>> A.nome = "Nilo Menezes" ❷
>>> A
<Classe Nome em 0x3140f10 Nome: Nilo Menezes Chave: nilo menezes>
>>> A.__nome ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Nome' object has no attribute '__nome'
>>> A.__chave ❹
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Nome' object has no attribute '__chave'
```

```
>>> A.chave ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Nome' object has no attribute 'chave'
>>> A._Nome__chave ❻
'nilo menezes'
```

Em ❶, criamos o objeto `A`, como anteriormente. Veja que o **`self.nome`** dentro do `__init__` chamou corretamente o método marcado por `@nome.setter`, e que `chave` e `nome` foram corretamente configuradas. Em ❷, alteramos o `nome` e logo podemos ver que a chave foi atualizada ao mesmo tempo. Observe que `A.nome` é acessível de fora da classe e que agora faz parte de nosso objeto como antes fazia o atributo **`self.nome`**. ❸, ❹ e ❺ mostram que `__nome`, `__chave` e `chave` são inacessíveis de fora da classe. Em ❻, mostramos o acesso à chave, com o truque de usar o nome completo após a escondida do Python (*name mangling*): `_nomedaclasse__atributo`, gerando: `_Nome__chave`. Entenda esse tipo de acesso como uma curiosidade e que, quando um programador marcar um atributo com `__`, está dizendo: não utilize esse atributo fora da classe, salvo se tiver certeza do que está fazendo.

O truque dos nomes que começam por `__` também funciona com métodos. Se você quiser marcar um método para ser utilizado apenas dentro da classe, adicione `__` antes de seu nome, da mesma forma que fizemos com nossos atributos.

Em todos os casos, nunca crie métodos ou atributos que comecem e terminem por `__`, esse tipo de construção é reservado aos métodos mágicos (especiais) da linguagem.

Você pode criar atributos que podem ser lidos definindo apenas o método de acesso com **`@property`**. Em nosso exemplo, `@nome.setter` é que permite alterarmos o nome. Se não utilizássemos `@nome.setter`, o nome seria acessível apenas para leitura, e uma exceção seria gerada se tentássemos alterar seu valor. Vejamos como isso funciona no Programa 10.2.

```
# Programa 10.2 - Chave como propriedade apenas para leitura (nome.py)
from functools import total_ordering
@total_ordering
class Nome:
    def __init__(self, nome):
        self.nome = nome
    def __str__(self):
        return self.nome
```

```
    def __repr__(self):
        return f"<Classe {type(self).__name__} em 0x{id(self):x} Nome: {self.__
    nome } Chave: {self.__chave}>"
    def __eq__(self, outro):
        return self.nome == outro.nome
    def __lt__(self, outro):
        return self.nome < outro.nome
    @property
    def nome(self):
        return self.__nome
    @nome.setter
    def nome(self, valor):
        if valor is None or not valor.strip():
            raise ValueError("Nome não pode ser nulo nem em branco")
        self.__nome = valor
        self.__chave = Nome.CriaChave(valor)
    @property
    def chave(self):
        return self.__chave
    @staticmethod
    def CriaChave(nome):
        return nome.strip().lower()
```

Veja que, no Programa 10.2, marcamos apenas o método com `@property` e não criamos um método com `@chave.setter` para processar as modificações. No interpretador, vejamos o que acontece se tentarmos alterar a chave:

```
>>> from nome import *
>>> A = Nome("Nilo")
>>> A.chave
'nilo'
>>> A.chave = "nilo menezes"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> A.nome = "Nilo Menezes"
>>> A.chave
'nilo menezes'
```

## 10.6 Revisitando a agenda

No Capítulo 9, desenvolvemos uma agenda de nomes e telefones. Agora que conhecemos classes, podemos adaptar o código da agenda de forma a melhorar seu funcionamento e a utilizar alguns conceitos de programação orientada a objetos.

Vamos revisitar a agenda de forma a definir suas estruturas como objetos e modelar a aplicação da agenda em uma classe separada. Nas seções anteriores, definimos as classes `Nome` e `ListaÚnica`. Essas classes serão utilizadas no modelo de dados de nossa nova agenda.

Podemos visualizar nossa agenda como uma lista de nomes e telefones. A agenda original do Capítulo 9 suportava apenas um telefone por nome, mas modificamos isso no Exercício 9.28, logo nossa nova agenda deverá suportar vários telefones, e cada telefone terá um tipo específico (celular, trabalho etc.).

Vamos começar pela classe que vai gerenciar os tipos de telefone. Vejamos o código:

```
from functools import total_ordering
@total_ordering
class TipoTelefone:
    def __init__(self, tipo):
        self.tipo = tipo
    def __str__(self):
        return f"({self.tipo })"
    def __eq__(self, outro):
        if outro is None:
            return False
        return self.tipo == outro.tipo
    def __lt__(self, outro):
        return self.tipo < outro.tipo
```

O programa implementa nosso tipo de telefone. Veja que implementamos o método `__str__` para exibir o nome do tipo entre parênteses, e o método `__eq__` e `__lt__` para ativar a comparação por tipo. Nesse exemplo, não usamos propriedades, pois na classe `TipoTelefone` não estamos fazendo qualquer verificação. Dessa forma, `self.tipo` oferecerá a mesma forma de acesso, seja como atributo ou propriedade. A grande vantagem de utilizar propriedades é que podemos, mais tarde, se necessário, transformar `self.tipo` em uma propriedade, sem alterar o código que utiliza a classe, como fizemos com o nome na classe `Nome`.

Observe que, no método `__eq__`, incluímos uma verificação se a outra parte for **None**. Essa verificação é necessária porque nossa futura agenda utilizará **None** quando não soubermos o tipo de um telefone. Assim, `self.tipo == outro.tipo` falharia, pois **None** não possui um atributo chamado `tipo`. O teste se outro for **None** foi feito com o operador **is** do Python. Isso é importante, pois, se utilizarmos ==, estaremos chamando recursivamente o método `__eq__`, que é chamado pelo operador de comparação (==).

A classe `Telefone` será implementada pelo programa a seguir:

```
class Telefone:
    def __init__(self, número, tipo=None):
        self.número = número
        self.tipo = tipo
    def __str__(self):
        if self.tipo is not None:
            tipo = self.tipo
        else:
            tipo = ""
        return f"{self.número} {tipo}"
    def __eq__(self, outro):
        return self.número == outro.número and (
              (self.tipo == outro.tipo) or (
               self.tipo is None or outro.tipo is None))
    @property
    def número(self):
        return self.__número
    @número.setter
    def número(self, valor):
        if valor is None or not valor.strip():
            raise ValueError("Número não pode ser None ou em branco")
        self.__número = valor
```

Na classe `Telefone`, fazemos o mesmo tipo de verificação de tipo que fizemos na classe `Nome`. Dessa forma, um objeto da classe `Telefone` não aceita números vazios. No método `__eq__`, implementamos a verificação de forma a ignorar um `Telefone` sem `TipoTelefone`, ou seja, um telefone em que o tipo é **None**. Dessa forma, se compararmos dois objetos de `Telefone` e um ou ambos possuírem o tipo valendo **None**, o resultado da comparação será decidido apenas se o número for idêntico. Se as duas instâncias de `Telefone` possuírem um tipo válido (diferente de **None**),

então o tipo fará parte da comparação, sendo iguais apenas se os números e os tipos forem iguais. Essas decisões foram tomadas para a implementação da agenda e são decisões de projeto. Isso significa que, para outras aplicações, essas decisões poderiam ser diferentes. O processo ficará mais claro quando lermos o programa completo da agenda.

Agora que temos as classes `Nome`, `Telefone` e `TipoTelefone`, podemos passar a pensar em como organizar os objetos dessas classes. Podemos ver nossa agenda como uma grande lista, em que cada elemento é um composto de `Nome` com uma lista de objetos do tipo `Telefone`. Vamos chamar cada entrada em nossa agenda de `DadoAgenda` e implementar uma classe para agrupar instâncias das duas classes:

```
import listaunica
class Telefones(ListaÚnica):
    def __init__(self):
        super().__init__(Telefone)
class DadoAgenda:
    def __init__(self, nome):
        self.nome = nome
        self.telefones = Telefones()
    @property
    def nome(self):
        return self.__nome
    @nome.setter
    def nome(self, valor):
        if not isinstance(type(valor), Nome):
            raise TypeError("nome deve ser uma instância da classe Nome")
        self.__nome = valor
    def pesquisaTelefone(self, telefone):
        posição = self.telefones.pesquisa(Telefone(telefone))
        if posição == -1:
            return None
        else:
            return self.telefones[posição]
```

Como cada objeto de `DadoAgenda` pode conter vários telefones, criamos uma classe `Telefones` que herda da classe `ListaÚnica` seu comportamento. Dessa forma `self.nome` de `DadoAgenda` é uma instância da classe `Nome`, e `self.telefones` uma instância de `Telefones`. Veja que adicionamos uma propriedade `nome` para facilitar o acesso ao objeto de `self.__nome` e fazer as verificações de tipo necessárias. Adicionamos também um método `pesquisaTelefone` que transforma uma string em um

objeto da classe Telefone, sem tipo. Esse objeto é então utilizado pelo método pesquisa, vindo da implementação original de ListaÚnica que retorna a posição do objeto na lista, ou -1, caso ele não seja encontrado. Veja que o método pesquisTelefone retorna uma instância de Telefone se ela for encontrada na lista ou **None**, caso contrário.

Vejamos agora a classe Agenda:

```
class TiposTelefone(ListaÚnica):
    def __init__(self):
        super().__init__(TipoTelefone)
class Agenda(ListaÚnica):
    def __init__(self):
        super().__init__(DadoAgenda)
        self.tiposTelefone = TiposTelefone()
    def adicionaTipo(self, tipo):
        self.tiposTelefone.adiciona(TipoTelefone(tipo))
    def pesquisaNome(self, nome):
        if isinstance(type(nome), str):
            nome = Nome(nome)
        for dados in self.lista:
            if dados.nome == nome:
                return dados
        else:
            return None
    def ordena(self):
        super().ordena(lambda dado: str(dado.nome))
```

O programa é uma listagem parcial, apenas para a explicação da classe Agenda. O programa completo precisa importar as definições de classe feitas anteriormente. Observe que a classe Agenda é uma subclasse de ListaÚnica configurada para aceitar somente objetos do tipo DadoAgenda. Veja que definimos uma classe TiposTelefone, também herdando de ListaÚnica, para manter a lista de tipos de telefone. Em nossa agenda, os tipos de telefone são pré-configurados na classe Agenda. Para facilitar o trabalho de inclusão de novos tipos, incluímos o método adicionaTipo, que prepara uma string, transformando-a em objeto de TipoTelefone e incluindo-o na lista de tipos válidos. Outro método a notar é pesquisaNome, que recebe o nome a pesquisar como objeto da classe Nome ou como string. Veja que utilizamos a verificação do tipo do parâmetro nome para transformá-lo em objeto de Nome, caso necessário, deixando nossa classe funcionar com strings ou com objetos do tipo Nome.

A função então pesquisa na lista interna de dados e retorna o objeto, caso encontrado. É importante notar a diferença dessa função de pesquisa e outras funções de pesquisa que definimos anteriormente. Em `pesquisaNome`, o objeto é retornado ou **`None`**, caso não seja encontrado. Da mesma forma que vimos o problema de referências em listas no Capítulo 6, vamos utilizar as referências retornadas por `pesquisaNome` para editar os valores da instância de `DadoAgenda` diretamente, sem a necessidade de reinseri-los na lista. Isso ficará mais claro quando analisarmos o programa da agenda completo. Para finalizar, o método `ordena` cria uma função para extrair a chave de ordenação de `DadoAgenda`; nesse caso, o `nome`.

Antes de passarmos para o programa da agenda, vamos construir uma classe `Menu` para exibir o menu principal:

```
class Menu:
    def __init__(self):
        self.opções = [["Sair", None]]
    def adicionaopção(self, nome, função):
        self.opções.append([nome, função])
    def exibe(self):
        print("====")
        print("Menu")
        print("====\n")
        for i, opção in enumerate(self.opções):
            print(f"[{i}] - {opção[0]}")
        print()
    def execute(self):
        while True:
            self.exibe()
            escolha = valida_faixa_inteiro("Escolha uma opção: ",
                                           0, len(self.opções)-1)
            if escolha == 0:
                break
            self.opções[escolha][1]()
```

Nosso menu é bem simples, adicionando a opção para "Sair" como valor padrão. O método `exibe` mostra o menu na tela, percorrendo a lista de opções. Já o método `execute` mostra continuamente o menu, pedindo uma escolha e chamando o método correspondente. Vejamos como utilizar a classe `Menu` na inicialização da classe `AppAgenda` no programa completo:

```
# Programa 10.3 - Listagem completa da nova agenda
import sys
import pickle
from functools import total_ordering
def nulo_ou_vazio(texto):
    return texto is None or not texto.strip()
def valida_faixa_inteiro(pergunta, início, fim, padrão=None):
    while True:
        try:
            entrada = input(pergunta)
            if nulo_ou_vazio(entrada) and padrão is not None:
                entrada = padrão
            valor = int(entrada)
            if início <= valor <= fim:
                return valor
        except ValueError:
            print(f"Valor inválido, favor digitar entre {início} e {fim}")
def valida_faixa_inteiro_ou_branco(pergunta, início, fim):
    while True:
        try:
            entrada = input(pergunta)
            if nulo_ou_vazio(entrada):
                return None
            valor = int(entrada)
            if início <= valor <= fim:
                return valor
        except ValueError:
            print(f"Valor inválido, favor digitar entre {início} e {fim}")
class ListaÚnica:
    def __init__(self, elem_class):
        self.lista = []
        self.elem_class = elem_class
    def __len__(self):
        return len(self.lista)
    def __iter__(self):
        return iter(self.lista)
    def __getitem__(self, p):
        return self.lista[p]
```

```
    def indiceVálido(self, i):
        return i >= 0 and i < len(self.lista)
    def adiciona(self, elem):
        if self.pesquisa(elem) == -1:
            self.lista.append(elem)
    def remove(self, elem):
        self.lista.remove(elem)
    def pesquisa(self, elem):
        self.verifica_tipo(elem)
        try:
            return self.lista.index(elem)
        except ValueError:
            return -1
    def verifica_tipo(self, elem):
        if not isinstance(type(elem), self.elem_class):
            raise TypeError("Tipo inválido")
    def ordena(self, chave=None):
        self.lista.sort(key=chave)
@total_ordering
class Nome:
    def __init__(self, nome):
        self.nome = nome
    def __str__(self):
        return self.nome
    def __repr__(self):
        return f"<Classe {type(self).__name__} em 0x{id(self):x} Nome: {self.__
    nome} Chave: {self.__chave }>"
    def __eq__(self, outro):
        return self.nome == outro.nome
    def __lt__(self, outro):
        return self.nome < outro.nome
    @property
    def nome(self):
        return self.__nome
    @nome.setter
    def nome(self, valor):
        if nulo_ou_vazio(valor):
            raise ValueError("Nome não pode ser nulo nem em branco")
        self.__nome = valor
```

```
            self.__chave = Nome.CriaChave(valor)
        @property
        def chave(self):
            return self.__chave
        @staticmethod
        def CriaChave(nome):
            return nome.strip().lower()
    @total_ordering
    class TipoTelefone:
        def __init__(self, tipo):
            self.tipo = tipo
        def __str__(self):
            return f"({self.tipo})"
        def __eq__(self, outro):
            if outro is None:
                return False
            return self.tipo == outro.tipo
        def __lt__(self, outro):
            return self.tipo < outro.tipo
    class Telefone:
        def __init__(self, número, tipo=None):
            self.número = número
            self.tipo = tipo
        def __str__(self):
            if self.tipo is not None:
                tipo = self.tipo
            else:
                tipo = ""
            return f"{self.número} {tipo}"
        def __eq__(self, outro):
            return self.número == outro.número and (
                   (self.tipo == outro.tipo) or (
                    self.tipo is None or outro.tipo is None))
        @property
        def número(self):
            return self.__número
        @número.setter
        def número(self, valor):
            if nulo_ou_vazio(valor):
```

```
            raise ValueError("Número não pode ser None ou em branco")
        self.__número = valor
class Telefones(ListaÚnica):
    def __init__(self):
        super().__init__(Telefone)
class TiposTelefone(ListaÚnica):
    def __init__(self):
        super().__init__(TipoTelefone)
class DadoAgenda:
    def __init__(self, nome):
        self.nome = nome
        self.telefones = Telefones()
    @property
    def nome(self):
        return self.__nome
    @nome.setter
    def nome(self, valor):
        if not isinstance(type(valor),Nome):
            raise TypeError("nome deve ser uma instância da classe Nome")
        self.__nome = valor
    def pesquisaTelefone(self, telefone):
        posição = self.telefones.pesquisa(Telefone(telefone))
        if posição == -1:
            return None
        else:
            return self.telefones[posição]
class Agenda(ListaÚnica):
    def __init__(self):
        super().__init__(DadoAgenda)
        self.tiposTelefone = TiposTelefone()
    def adicionaTipo(self, tipo):
        self.tiposTelefone.adiciona(TipoTelefone(tipo))
    def pesquisaNome(self, nome):
        if isinstance(type(nome), str):
            nome = Nome(nome)
        for dados in self.lista:
            if dados.nome == nome:
                return dados
        else:
```

```
            return None
    def ordena(self):
        super().ordena(lambda dado: str(dado.nome))
class Menu:
    def __init__(self):
        self.opções = [["Sair", None]]
    def adicionaopção(self, nome, função):
        self.opções.append([nome, função])
    def exibe(self):
        print("====")
        print("Menu")
        print("====\n")
        for i, opção in enumerate(self.opções):
            print(f"[{i}] - {opção[0]}")
        print()
    def execute(self):
        while True:
            self.exibe()
            escolha = valida_faixa_inteiro("Escolha uma opção: ",
                                           0, len(self.opções) - 1)
            if escolha == 0:
                break
            self.opções[escolha][1]()
class AppAgenda:
    @staticmethod
    def pede_nome():
        return input("Nome: ")
    @staticmethod
    def pede_telefone():
        return input("Telefone: ")
    @staticmethod
    def mostra_dados(dados):
        print(f"Nome: {dados.nome}")
        for telefone in dados.telefones:
            print(f"Telefone: {telefone}")
        print()
    @staticmethod
    def mostra_dados_telefone(dados):
        print(f"Nome: {dados.nome}")
```

```
        for i, telefone in enumerate(dados.telefones):
            print(f"{i} - Telefone: {telefone}")
        print()
    @staticmethod
    def pede_nome_arquivo():
        return input("Nome do arquivo: ")
    def __init__(self):
        self.agenda = Agenda()
        self.agenda.adicionaTipo("Celular")
        self.agenda.adicionaTipo("Residência")
        self.agenda.adicionaTipo("Trabalho")
        self.agenda.adicionaTipo("Fax")
        self.menu = Menu()
        self.menu.adicionaopção("Novo", self.novo)
        self.menu.adicionaopção("Altera", self.altera)
        self.menu.adicionaopção("Apaga", self.apaga)
        self.menu.adicionaopção("Lista", self.lista)
        self.menu.adicionaopção("Grava", self.grava)
        self.menu.adicionaopção("Lê", self.lê)
        self.menu.adicionaopção("Ordena", self.ordena)
        self.ultimo_nome = None
    def pede_tipo_telefone(self, padrão=None):
        for i, tipo in enumerate(self.agenda.tiposTelefone):
            print(f" {i} - {tipo} ", end=None)
        t = valida_faixa_inteiro("Tipo: ", 0, len(self.agenda.tiposTelefone) - 1,
padrão)
        return self.agenda.tiposTelefone[t]
    def pesquisa(self, nome):
         dado = self.agenda.pesquisaNome(nome)
         return dado
    def novo(self):
        novo = AppAgenda.pede_nome()
        if nulo_ou_vazio(novo):
            return
        nome = Nome(novo)
        if self.pesquisa(nome) is not None:
            print("Nome já existe!")
            return
        registro = DadoAgenda(nome)
```

```
        self.menu_telefones(registro)
        self.agenda.adiciona(registro)
    def apaga(self):
        if len(self.agenda) == 0:
            print("Agenda vazia, nada a apagar")
        nome = AppAgenda.pede_nome()
        if nulo_ou_vazio(nome):
           return
        p = self.pesquisa(nome)
        if p is not None:
            self.agenda.remove(p)
            print(f"Apagado. A agenda agora possui apenas: {len(self.agenda)} registros")
        else:
            print("Nome não encontrado.")
    def altera(self):
        if len(self.agenda) == 0:
            print("Agenda vazia, nada a alterar")
        nome = AppAgenda.pede_nome()
        if nulo_ou_vazio(nome):
           return
        p = self.pesquisa(nome)
        if p is not None:
            print("\nEncontrado:\n")
            AppAgenda.mostra_dados(p)
            print("Digite enter caso não queira alterar o nome")
            novo = AppAgenda.pede_nome()
            if not nulo_ou_vazio(novo):
                p.nome = Nome(novo)
            self.menu_telefones(p)
        else:
            print("Nome não encontrado!")
    def menu_telefones(self, dados):
        while True:
            print("\nEditando telefones\n")
            AppAgenda.mostra_dados_telefone(dados)
            if len(dados.telefones) > 0:
                print("\n[A] - alterar\n[D] - apagar\n", end="")
            print("[N] - novo\n[S] - sair\n")
            operação = input("Escolha uma operação: ")
```

```
            operação = operação.lower()
            if operação not in ["a","d","n", "s"]:
                print("Operação inválida. Digite A, D, N ou S")
                continue
            if operação == 'a' and len(dados.telefones) > 0:
                self.altera_telefones(dados)
            elif operação == 'd' and len(dados.telefones) > 0:
                self.apaga_telefone(dados)
            elif operação == 'n':
                self.novo_telefone(dados)
            elif operação == "s":
                break
    def novo_telefone(self, dados):
        telefone = AppAgenda.pede_telefone()
        if nulo_ou_vazio(telefone):
            return
        if dados.pesquisaTelefone(telefone) is not None:
            print("Telefone já existe")
        tipo = self.pede_tipo_telefone()
        dados.telefones.adiciona(Telefone(telefone, tipo))
    def apaga_telefone(self, dados):
        t = valida_faixa_inteiro_ou_branco(
            "Digite a posição do número a apagar, enter para sair: ",
            0, len(dados.telefones) - 1)
        if t is None:
            return
        dados.telefones.remove(dados.telefones[t])
    def altera_telefones(self, dados):
        t = valida_faixa_inteiro_ou_branco(
            "Digite a posição do número a alterar, enter para sair: ",
            0, len(dados.telefones)-1)
        if t is None:
            return
        telefone = dados.telefones[t]
        print(f"Telefone: {telefone}")
        print("Digite enter caso não queira alterar o número")
        novotelefone = AppAgenda.pede_telefone()
        if not nulo_ou_vazio(novotelefone):
            telefone.número = novotelefone
```

```
            print("Digite enter caso não queira alterar o tipo")
            telefone.tipo = self.pede_tipo_telefone(
                self.agenda.tiposTelefone.pesquisa(telefone.tipo))
        def lista(self):
            print("\nAgenda")
            print("-" * 60)
            for e in self.agenda:
                AppAgenda.mostra_dados(e)
            print("-" * 60)
        def lê(self, nome_arquivo=None):
            if nome_arquivo is None:
                nome_arquivo = AppAgenda.pede_nome_arquivo()
            if nulo_ou_vazio(nome_arquivo):
                return
            with open(nome_arquivo, "rb") as arquivo:
                self.agenda = pickle.load(arquivo)
            self.ultimo_nome = nome_arquivo
        def ordena(self):
            self.agenda.ordena()
            print("\nAgenda ordenada\n")
        def grava(self):
            if self.ultimo_nome is not None:
                print(f"Último nome utilizado foi '{self.ultimo_nome}'")
                print("Digite enter caso queira utilizar o mesmo nome")
            nome_arquivo = AppAgenda.pede_nome_arquivo()
            if nulo_ou_vazio(nome_arquivo):
                if self.ultimo_nome is not None:
                    nome_arquivo = self.ultimo_nome
                else:
                    return
            with open(nome_arquivo, "wb") as arquivo:
                pickle.dump(self.agenda, arquivo)
        def execute(self):
            self.menu.execute()
    if __name__ == "__main__":
        app = AppAgenda()
        if len(sys.argv) > 1:
            app.lê(sys.argv[1])
        app.execute()
```

Todas as classes necessárias foram escritas em um só arquivo para facilitar a leitura. A classe `AppAgenda` é nossa aplicação agenda em si. Veja a implementação de `__init__`, na qual criamos uma instância de `Agenda` para conter nossos dados e populamos os tipos de telefone que queremos trabalhar. Também criamos uma instância de `Menu` e adicionamos as opções. Observe que passamos o nome da opção e o método correspondente a cada escolha. O atributo `self.ultimo_nome` é usado para guardar o nome usado na última leitura do arquivo.

Na nova agenda, note que dividimos o tratamento de nomes e telefones. Como podemos ter vários telefones, outro menu com opções para o gerenciamento de telefones aparece. Você pode ver os detalhes na implementação do método `menu_telefones`. Em nossa agenda, adotamos o seguinte comportamento quando um valor não muda, digitamos apenas **Enter**. Em todas as opções, você deve perceber o tratamento de entradas em branco, até criamos uma função de suporte para isso: `nulo_ou_vazio`.

Em todos os métodos de edição, você pode perceber que utilizamos apenas os métodos fornecidos pela classe `ListaÚnica`. Veja também que as pesquisas retornam os objetos, e não apenas a posição deles na lista.

Os métodos `grava` e `lê` foram modificados para utilizar o módulo `pickle` do Python. Como nossa agenda agora possui vários telefones, cada um com um tipo, a utilização de arquivos simples se tornaria muito trabalhosa. O módulo `pickle` fornece métodos que gravam um objeto ou uma lista de objetos em disco. No método `grava`, utilizamos a função `pickle.dump`, que grava no arquivo o objeto agenda inteiro. No método `lê`, utilizamos `pickle.load` para recriar nossa agenda a partir do arquivo em disco. A utilização do `pickle` facilita muito a tarefa de serializar os dados em um arquivo, ou seja, representar os objetos de forma que possam ser reconstruídos, caso necessário (desserialização).

Como última modificação, adicionamos a possibilidade de passar o nome do arquivo da agenda como parâmetro. Caso o nome seja passado, o arquivo é lido antes de apresentar o menu. Observe também que o método `execute` tem um loop infinito que roda até sairmos do menu principal.

## 10.7 Criando exceções

Para facilitar o gerenciamento de exceções em nossos programas, podemos criar novos tipos de exceção para diferenciar os erros gerados pela nossa aplicação dos gerados pela linguagem ou suas bibliotecas. Para criar nossas próprias exceções, vamos utilizar herança (ver Seção 10.4) para estender a classe **`Exception`** do Python.

```
class NovaException(Exception):
    pass

def lançador():
    raise NovaException

try:
    lançador()
except NovaException as n:
    print("Uma exceção do tipo NovaException foi lançada")
```

Veja que `NovaException` é uma subclasse de **`Exception`**. Ao definirmos nossas próprias exceções, estas devem derivar da classe **`Exception`**. Observe que podemos utilizar essa nova exceção como todas as outras em Python, por exemplo, em um bloco **`try except`**. No caso, `NovaException` não adiciona nenhum método ou atributo à classe **`Exception`**, atuando apenas como um novo tipo, por isso, utilizamos apenas o **`pass`** em sua definição. A instrução **`pass`** pode ser utilizada em Python para indicar o fim de um bloco que seria vazio. Ela também pode ser usada para definir funções vazias quando ainda estamos escrevendo nosso programa e queremos apenas satisfazer a sintaxe da linguagem, para mais tarde modificar ou adicionar conteúdo e remover **`pass`**.

Ao desenvolver novos programas, é recomendado que você crie um tipo próprio de exceção para servir de super classe a todas as exceções do programa. Essas exceções não precisam substituir as previamente definidas pela linguagem, mas definir um significado próprio ao programa. Um exemplo de exceção própria a um programa é a tentativa de retirar mais dinheiro que o disponível na conta, no exemplo de nosso banco. Uma exceção do tipo `SaldoIndisponível` faz mais sentido que uma mera **`Exception`** e facilita o tratamento de erros.

```
class BancoException(Exception):
    pass

class SaldoIndisponível(BancoException):
    pass

class ClienteNãoExiste(BancoException):
    pass
```

```
def saque(saldo, valor):
    if valor > saldo:
        raise SaldoIndisponível
    return saldo - valor

try:
    saldo = saque(100, 500)
except SaldoIndisponível:
    print("Erro: Saldo insuficiente")
```

Podemos também criar novas exceções que adicionem informações ou atributos.

```
class EstoqueException(Exception):
    def __init__(self, mensagem, codigo_de_erro):
        super().__init__(mensagem)
        self.codigo_de_erro = codigo_de_erro

def verifique_quantidade(quantidade):
    if quantidade < 0:
        raise EstoqueException("Quantidade negativa", codigo_de_erro=1)


try:
    verifique_quantidade(-10)
except EstoqueException as ee:
    print(f"Erro: {ee.codigo_de_erro} {ee}")
```

# CAPÍTULO 11

# Banco de dados

Praticamente todo programa precisa ler ou armazenar dados. Para uma quantidade pequena de informação, arquivos simples resolvem o problema, mas, uma vez que os dados precisem ser atualizados, problemas de manutenção e dificuldade de acesso começam a aparecer.

Depois de algum tempo, o programador começa a ver que as operações realizadas com arquivos seguem alguns padrões, como: inserir, alterar, apagar e pesquisar. Ele também começa a perceber que as consultas podem ser feitas com critérios diferentes e que, à medida que o arquivo cresce, as operações se tornam mais lentas. Todos esses tipos de problema foram identificados e resolvidos há bastante tempo, com o uso de programas gerenciadores de banco de dados.

Programas gerenciadores de banco de dados foram desenvolvidos de forma a organizar e facilitar o acesso a grandes massas de informação. No entanto, para usarmos um banco de dados, precisamos saber como eles são organizados. Neste capítulo, abordaremos os conceitos básicos de banco de dados, bem como a utilização da linguagem SQL e o acesso via linguagem de programação, no caso, Python.

Os programas do Capítulo 11 são bem maiores que os do restante do livro. É recomendado ler este capítulo perto de um computador e em alguns casos com os programas impressos em papel ou facilmente acessíveis no site do livro.

## 11.1 Conceitos básicos

Para começarmos a utilizar os termos de banco de dados, é importante entender como as coisas funcionavam antes de usarmos computadores para controlar o registro de informações.

Não muito tempo atrás, as pessoas usavam a própria memória para armazenar informações importantes, como os números de telefones de amigos próximos e

até o próprio número de telefone. Hoje, com a proliferação de celulares, praticamente perdemos essa capacidade, por falta de uso. Mas, para entender banco de dados, precisamos compreender por que eles foram criados, quais problemas eles resolveram e como eram as coisas antes deles.

Imagine que você é uma pessoa extremamente popular e que precisa controlar mais de 100 contatos (nome e telefone) dos amigos mais próximos. Lembre-se de imaginar essa situação num mundo sem computadores. O que você faria para controlar todos os dados?

Provavelmente, você escreveria todos os nomes e telefones de seus novos contatos em folhas de papel. Usando um caderno, poder-se-ia facilmente anotar um nome abaixo do outro, com o número de telefone anotado mais à direita da folha.

*Tabela 11.1 – Exemplo de nomes anotados numa folha de papel*

| Nome | Telefone |
|---|---|
| João | 98901-0109 |
| André | 98902-8900 |
| Maria | 97891-3321 |

O problema com anotações em papel é que não conseguimos alterar os dados lá escritos, salvo se escrevermos a lápis, mas o resultado nunca é muito agradável. Outro problema é que não conhecemos novas pessoas em ordem alfabética, o que resulta numa lista de nomes e telefones desordenada. O papel também não ajuda quando temos mais amigos com nomes que começam por uma letra que outra, ou quando nossos amigos têm vários nomes, como João Carlos, e você nunca se lembra se registrou como João ou como Carlos!

Agora, imagine que estamos controlando não apenas nossos amigos, mas também contatos comerciais. No caso dos amigos, apenas nome e telefone já bastam para reestabelecer o contato. No caso de um contato comercial, a empresa onde a pessoa trabalha e a função que ela desempenha são importantes também. Com o tempo, precisaríamos de vários cadernos ou pastas, um para cada tipo de contato, organizados em armários, talvez uma gaveta para cada tipo de registro. Antes de os computadores se tornarem populares, as coisas eram organizadas dessa forma, e em muitos lugares são assim até hoje.

Uma vez que os problemas que podem ser resolvidos com banco de dados foram apresentados, nós já podemos aprender alguns conceitos importantíssimos para continuar o estudo, como: campos, registros, tabelas e tipos de dados.

Campos são a menor unidade de informação em um sistema gerenciador de banco de dados. Se fizermos uma comparação com nossa agenda no papel, nome e telefone seriam dois campos. O campo nome armazenaria o nome de cada contato; e o campo telefone, o número de telefone, respectivamente.

Cada linha de nossa agenda seria chamada de registro. Um registro é formado por um conjunto conhecido de campos. Em nosso exemplo, cada pessoa na agenda, com seu nome e telefone, formaria um registro.

Podemos pensar em tabelas do banco de dados como a unidade de armazenamento de registros do mesmo tipo. Imagine uma entrada da agenda telefônica, onde cada registro, contendo nome e telefone, seria armazenado. O conjunto de registros do mesmo tipo é organizado em tabelas, nesse caso, na tabela agenda ou lista telefônica.

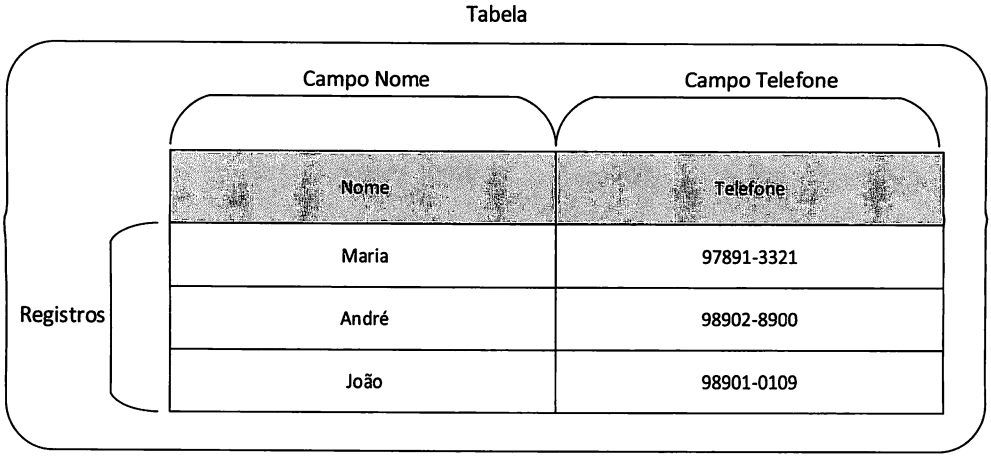


*Figura 11.1 – Campos, registros e tabela.*

Os conceitos de campo, registro e tabela são fundamentais para o entendimento do resto de texto, vide Figura 11.1. Não hesite em reler esta seção, caso algum desses conceitos ainda não estejam claros para você. Eles serão reapresentados nas seções seguintes, quando serão aplicados em um banco de dados demonstrativo.

## 11.2 SQL

*Structured Query Language* (SQL – Linguagem de Consulta Estruturada) é a linguagem usada para criar bancos de dados, gerar consultas, manipular (inserir, atualizar, alterar e apagar) registros e, principalmente, realizar consultas. É uma linguagem de programação especializada na manipulação de dados, baseada na

álgebra relacional e no modelo relacional criado por Edgar F. Codd (http://pt.wikipedia.org/wiki/Edgar_Frank_Codd).

Neste capítulo, nós veremos como escrever comandos SQL para o banco SQLite que vem pré-instalado com o interpretador Python e que é facilmente acessível de um programa. A linguagem SQL é definida por vários padrões, como SQL-92, mas cada banco de dados introduz modificações e adições ao padrão, embora o funcionamento básico continue o mesmo. Neste capítulo, nós veremos exclusivamente os comandos SQL no formato aceito pelo banco SQLite.

## 11.3 Python & SQLite

O SQLite é um gerenciador de banco de dados leve e completo, muito utilizado e presente mesmo em telefones celulares. Uma de suas principais características é não precisar de um servidor dedicado, sendo capaz de se iniciar a partir de seu programa. Nesta seção, nós veremos os comandos mais importantes e as etapas necessárias para utilizar o SQLite. Vejamos um programa Python que cria um banco de dados, uma tabela e um registro:

```
import sqlite3 ❶
conexão = sqlite3.connect("agenda.db") ❷
cursor = conexão.cursor() ❸
cursor.execute('''
        create table agenda(
            nome text,
            telefone text)
        ''') ❹
cursor.execute('''
        insert into agenda (nome, telefone)
            values(?, ?)
            ''', ("Nilo", "7788-1432")) ❺
conexão.commit() ❹
cursor.close() ❺
conexão.close() ❻
```

A primeira coisa a fazer é informar que utilizaremos um banco SQLite. Isso é feito em ❶. Depois do `import`, várias funções e objetos que acessam o banco de dados se tornam disponíveis ao seu programa. Antes de continuarmos, vamos criar o banco de dados em ❷. A conexão com o banco de dados se assemelha à manipulação de um arquivo, é a operação análoga a abrir um arquivo. O nome do banco de dados que estamos criando será gravado no arquivo `agenda.db`.

A extensão `.db` é apenas uma convenção, mas é recomendado diferenciar o nome do arquivo de um arquivo normal, principalmente porque todos os seus dados serão guardados nesse arquivo. A grande vantagem de um banco de dados é que o registro de informações e toda a manutenção dos dados são feitos automaticamente para você com comandos SQL.

Em ❸, criamos um cursor. Cursores são objetos utilizados para enviar comandos e receber resultados do banco de dados. Um cursor é criado para uma conexão, chamando-se o método `cursor()`. Uma vez que obtivemos um cursor, nós podemos enviar comandos ao banco de dados. O primeiro deles é criar uma tabela para guardar nomes e telefones. Vamos chamá-la de agenda:

```
create table agenda(nome text, telefone text)
```

O comando SQL usado para criar uma tabela é `create table`. Esse comando precisa do nome da tabela a criar; nesse exemplo, `agenda` e uma lista de campos entre parênteses. `Nome` e `telefone` são nossos campos e `text` é o tipo. Embora em Python não precisemos declarar o tipo de uma variável, a maioria dos bancos de dados exige um tipo para cada campo. No caso do SQLite, o tipo não é exigido, mas vamos continuar a usá-lo para que você não tenha problemas com outros bancos, e para que a noção de tipo comece a fazer sentido. Um campo do tipo `text` pode armazenar dados como uma string do Python.

Em ❹, utilizamos o método `execute` de nosso cursor para enviar o comando ao banco de dados. Observe que escrevemos o comando em várias linhas, usando apóstrofos triplos do Python. A linguagem SQL não exige essa formatação, embora ela deixe o comando mais claro e simples de entender. Você poderia ter escrito tudo em uma só linha e mesmo utilizar uma string simples do Python.

Com a tabela criada, podemos começar a introduzir nossos dados. Vejamos o comando SQL usado para inserir um registro:

```
insert into agenda (nome, telefone) values (?, ?)
```

O comando `insert` precisa do nome da tabela, na qual vamos inserir os dados, e também do nome dos campos e seus respectivos valores. `into` faz parte do comando `insert` e é escrito antes do nome da tabela. O nome dos campos é escrito logo a seguir, separados por vírgula e, dessa vez, não precisamos mais informar o tipo dos campos, apenas a lista de nomes. Os valores que vamos inserir na tabela são especificados também entre parênteses, mas na segunda parte do comando `insert` que começa após a palavra `values`. Em nosso exemplo, a posição de cada valor foi marcada com interrogações, uma para cada campo. A ordem dos valores é a mesma dos campos; logo, a primeira interrogação se refere ao campo `nome`; a segunda, ao campo `telefone`.

A linguagem SQL permite que escrevamos os valores diretamente no comando, como uma grande string, mas, hoje em dia, esse tipo de sintaxe não é recomendado, por ser inseguro e facilmente utilizado para gerar um ataque de segurança de dados chamado SQLInjection (`http://pt.wikipedia.org/wiki/Inje%C3%A7%C3%A3o_de_SQL`). Você não precisa se preocupar com isso agora, principalmente porque, ao utilizarmos as interrogações, estamos utilizando parâmetros que evitam esse tipo de problema. Podemos entender as interrogações como um equivalente das máscaras de string do Python, mas que utilizaremos com comandos SQL.

Em ❺, utilizamos o método `execute` para executar o comando `insert`, mas, dessa vez, passamos os dados logo após o comando. No exemplo, "Nilo" e "7788-1432" vão substituir a primeira e a segunda interrogação quando o comando for executado. É importante notar que os dois valores foram passados como uma tupla.

Uma vez que o comando é executado, os dados são enviados para o banco de dados, mas ainda não estão gravados definitivamente. Isso acontece, pois estamos usando uma transação. Transações serão apresentadas com mais detalhes em outra seção; por enquanto, considere o comando `commit` em ❻ como parte das operações necessárias para modificar o banco de dados.

Antes de terminarmos o programa, fechamos (`close`) o cursor e a conexão com o banco de dados, respectivamente em ❼ e ❽. Veremos mais adiante como usar a sentença **`with`** do Python para facilitar essas operações.

Execute o programa e verifique se o arquivo `agenda.db` foi criado. Se você executar o programa uma segunda vez, um erro será gerado com a mensagem:

```
Traceback (most recent call last):
  File "criatabela.py", line 9, in <module>
    ''')
sqlite3.OperationalError: table agenda already exists
```

Esse erro acontece porque a tabela `agenda` já existe. Se você precisar executar o programa novamente, apague o arquivo `agenda.db`. Lembre-se de que todos os dados estão nesse arquivo e, ao apagá-lo, tudo é perdido. Você pode apagar esse arquivo sempre que quiser reinicializar o banco de dados.

Vejamos agora como ler os dados que gravamos no banco de dados, vamos fazer uma consulta (*query*). O programa a seguir realiza a consulta e mostra os resultados na tela:

```
import sqlite3
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
```

```
cursor.execute("select * from agenda") ❶
resultado = cursor.fetchone() ❷
print(f"Nome: {resultado[0]}\nTelefone: {resultado[1]}") ❸
cursor.close()
conexão.close()
```

O programa é muito parecido com o anterior, uma vez que precisamos importar o módulo do SQLite, estabelecer uma conexão e criar um cursor. O comando SQL que realiza uma consulta é o comando `select`.

```
select * from agenda
```

O comando `select`, em sua forma mais simples, utiliza uma lista de campos e uma lista de tabelas. Em nosso exemplo, a lista de campos foi substituída por `*` (asterisco). O asterisco representa todos os campos da tabela sendo consultada, nesse caso `nome` e `telefone`. A palavra `from` é utilizada para separar a lista de campos da lista de tabelas. Em nosso exemplo, apenas a tabela `agenda`. O comando `select` é executado na linha ❶.

Para acessar os resultados do comando `select`, devemos utilizar o método `fetchone` de nosso cursor ❷. Esse método retorna uma tupla com os resultados de nossa consulta ou **`None`**, caso a tabela esteja vazia. Para simplificar nosso exemplo, o teste de **`None`** foi retirado.

A tupla retornada possui a mesma ordem dos campos de nossa consulta, nesse caso `nome` e `telefone`. Assim, `resultado[0]` é o primeiro campo, no caso `nome`, e `resultado[1]` é o segundo, `telefone`. Em ❸, usamos uma f-string em Python e uma máscara que usa os campos da tupla `resultado`.

Execute o programa e verifique o resultado:

```
Nome: Nilo
Telefone: 7788-1432
```

Vejamos agora como incluir os outros telefones de nossa agenda. O programa seguinte apresenta o método `executemany`. A principal diferença entre `executemany` e `execute` é que `executemany` trabalha com vários valores. Em nosso exemplo, utilizamos uma lista de tuplas, `dados`. Cada elemento da lista é uma tupla com dois valores, exatamente como fizemos no programa anterior.

```
import sqlite3
dados = [("João",  "98901-0109"),
         ("André", "98902-8900"),
         ("Maria", "97891-3321")]
conexão = sqlite3.connect("agenda.db")
```

```
cursor = conexão.cursor()
cursor.executemany('''
      insert into agenda (nome, telefone)
      values(?, ?)
      ''', dados)
conexão.commit()
cursor.close()
conexão.close()
```

Com os dados inseridos pelo programa, nossa agenda deve ter agora 4 registros. Vejamos como imprimir o conteúdo de nossa tabela, usando o mesmo comando SQL, mas, dessa vez, trabalhando com vários resultados.

```
# Programa 11.1 - Consulta com múltiplos resultados
import sqlite3
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute("select * from agenda")
resultado = cursor.fetchall() ❶
for registro in resultado:
    print(f"Nome: {registro[0]}\nTelefone: {registro[1]}") ❷
cursor.close()
conexão.close()
```

Em ❶, utilizamos o método `fetchall` de nosso cursor para retornar uma lista com os resultados de nossa consulta. Em ❷, utilizamos a variável `registro` para exibir os dados. Assim como vimos o método `executemany`, que aceita uma lista de tuplas como parâmetro, `fetchall` retorna uma lista de tuplas. Cada elemento dessa lista é uma tupla contendo todos os campos retornados pela consulta. Uma vez que temos a lista `resultado`, utilizamos um simples **`for`** para trabalhar com cada registro.

O método `fetchall` retorna **`None`** caso o resultado da consulta seja vazio. Veremos isso em outros exemplos. Para consultas pequenas, contendo poucos registros como resultado, o método `fetchall` é muito interessante e fácil de utilizar. Para consultas maiores, em que mais de 100 registros são retornados, outros métodos de obter os resultados da consulta podem ser mais interessantes. Esses métodos evitam a criação de uma longa lista, que pode ocupar uma grande quantidade de memória e demorar muito tempo para executar.

O programa seguinte mostra o método `fetchone` ❶ sendo utilizado dentro de uma estrutura de repetição **`while`**. Como não sabemos quantos registros serão retornados, utilizamos um **`while True`**, que é interrompido quando o método

fetchone retorna **None**, significando que todos os resultados da consulta já foram obtidos. Você pode ler *fetch* como obter; logo, fetchone seria obter um resultado e fetchall obter todos os resultados. A vantagem de fetchone nesse caso é que imprimimos o resultado da consulta tão logo obtemos um e mantemos a impressão à medida que outros resultados forem chegando. Esse tempo de entrega é um conceito importante a perceber, uma vez que os dados vêm do banco de dados para o nosso programa. Essa transferência é controlada pelo banco de dados, responsável por executar nossa consulta e gerar os resultados.

```
# Programa 11.2 - Consulta, registro por registro
import sqlite3
conexão = sqlite3.connect("agenda.db")
cursor = conexão.cursor()
cursor.execute("select * from agenda")
while True:
    resultado = cursor.fetchone() ❶
    if resultado is None:
        break
    print(f"Nome: {resultado[0]}\nTelefone: {resultado[1]}")
cursor.close()
conexão.close()
```

Antes de passarmos para comandos SQL mais avançados, vejamos a estrutura **with** do Python que pode nos ajudar a não nos esquecermos de chamar os métodos close de nossos objetos.

O Programa 11.3 mostra o programa equivalente ao Programa 11.2, mas utilizando a cláusula **with**. Uma das vantagens de utilizarmos **with** é que criamos um bloco onde um objeto é tido como válido. Se algo acontecer dentro do bloco, como uma exceção, a estrutura **with** garante que o método close será chamado. Na realidade, **with** chama o método __exit__ no fim do bloco e funciona muito bem com arquivos e conexões de banco de dados. Infelizmente, cursores não possuem o método __exit__, obrigando-nos a chamar manualmente o método close, ou a importar um módulo especial, contextlib, que oferece a função closing ❶, que adapta um cursor com um método __exit__, que chama close. Por enquanto, esse detalhe pode ficar apenas como uma curiosidade, mas falaremos mais de **with** no restante deste capítulo.

```
# Programa 11.3 - Uso do with para fechar a conexão
import sqlite3
from contextlib import closing ❶
```

```
with sqlite3.connect("agenda.db") as conexão:
    with closing(conexão.cursor()) as cursor:
        cursor.execute("select * from agenda")
        while True:
            resultado = cursor.fetchone()
            if resultado is None:
                break
            print(f"Nome: {resultado[0]}\nTelefone: {resultado[1]}")
```

**Exercício 11.1** Faça um programa que crie o banco de dados *preços.db* com a tabela preços para armazenar uma lista de preços de venda de produtos. A tabela deve conter o nome do produto e seu respectivo preço. O programa também deve inserir alguns dados para teste.

**Exercício 11.2** Faça um programa para listar todos os preços do banco *preços.db*.

## 11.4 Consultando registros

Até agora, não fomos além do que poderíamos ter feito com simples arquivos texto. A facilidade de um sistema de banco de dados começa a aparecer quando precisamos procurar e alterar dados. Ao trabalharmos com arquivos, essas operações devem ser implementadas em nossos programas, mas, com o SQLite, podemos realizá-las usando comandos SQL. Primeiro, vamos utilizar uma variação do comando select para mostrar apenas alguns registros, implementando uma seleção de registros com base em uma pesquisa. Pesquisas em SQL são feitas com a cláusula where. Vejamos o comando SQL que seleciona todos os registros da agenda, cujo nome seja igual a "Nilo".

```
select * from agenda where nome = "Nilo"
```

Veja que apenas acrescentamos a cláusula where após o nome da tabela. O critério de seleção ou de pesquisa deve ser escrito como uma expressão, no caso nome = "Nilo". O Programa 11.4 mostra o programa 0com essa modificação.

```
# Programa 11.4 - Consulta com filtro de seleção
import sqlite3
from contextlib import closing

with sqlite3.connect("agenda.db") as conexão:
    with closing(conexão.cursor()) as cursor:
```

```
        cursor.execute("select * from agenda where nome = 'Nilo'")
        while True:
            resultado = cursor.fetchone()
            if resultado is None:
                break
        print(f"Nome: {resultado[0]}\nTelefone: {resultado[1]}")
```

Ao executarmos o Programa 11.4, devemos ter apenas um resultado:

```
Nome: Nilo
Telefone: 7788-1432
```

Veja que escrevemos `'Nilo'` entre apóstrofos. Aqui, podemos usar um pouco do que já sabemos sobre strings em Python e escrever:

```
cursor.execute('select * from agenda where nome = "Nilo"')
```

Ou seja, poderíamos trocar as aspas por apóstrofos ou ainda usar aspas triplas:

```
cursor.execute("""select * from agenda where nome = "Nilo" """)
```

No caso de nosso exemplo, o nome 'Nilo' é uma constante e não há problemas em escrevê-lo diretamente em nosso comando `select`. No entanto, caso o nome a filtrar viesse de uma variável, ficaríamos tentados a escrever um programa, como:

```
import sqlite3
from contextlib import closing

nome = input("Nome a selecionar: ")
with sqlite3.connect("agenda.db") as conexão:
    with closing(conexão.cursor()) as cursor:
        cursor.execute(f'select * from agenda where nome = "{nome}"')
        while True:
            resultado = cursor.fetchone()
            if resultado is None:
                break
print(f"Nome: {resultado[0]}\nTelefone: {resultado[1]}")
```

Execute o programa com vários valores: Nilo, João e Maria. Experimente também com um nome que não existe. A cláusula `where` funciona de forma parecida a um filtro. Imagine que o comando `select` cria uma lista e que a expressão lógica definida no `where` é avaliada para cada elemento. Quando o resultado dessa avaliação é verdadeiro, a linha é copiada para uma outra lista, a lista de resultados, retornada pela nossa consulta.

Veja que o programa funciona relativamente muito bem, exceto quando nada encontramos e o programa termina sem dizer muita coisa. Nós vamos corrigir esse problema logo a seguir, mas execute o programa mais uma vez e digite a seguinte sequência como nome:

```
X" or "1"="1
```

Surpreso com o resultado? Esse é o motivo de não utilizarmos variáveis em nossas consultas. Esse tipo de vulnerabilidade é um exemplo de SQLInjection, um ataque bem conhecido. Isso acontece porque o comando SQL resultante é:

```
select * from agenda where nome = "X" or "1"="1"
```

Para evitar esse tipo de ataque, sempre utilize parâmetros com valores variáveis.

O or da linguagem SQL funciona de forma semelhante ao **or** do Python. Dessa forma, nossa entrada de dados foi modificada por um valor digitado no programa. Esse tipo de erro é muito grave e pode ficar muito tempo em nossos programas sem ser percebido. Isso acontece porque a consulta é uma string como outra qualquer, e o valor passado para o método execute é a string resultante. Dessa forma, o valor digitado pelo usuário pode introduzir elementos que nós não desejamos. Os operadores relacionais **and** e **not** funcionam exatamente como em Python, e você também pode usá-los em expressões SQL.

Para não cairmos nesse tipo de armadilha, utilizaremos sempre parâmetros em nossas consultas:

```
# Programa 11.5 - Consulta utilizando parâmetros
import sqlite3
from contextlib import closing
nome = input("Nome a selecionar: ")
with sqlite3.connect("agenda.db") as conexão:
    with closing(conexão.cursor()) as cursor:
        cursor.execute('select * from agenda where nome = ?', (nome,))
        x = 0
        while True:
            resultado = cursor.fetchone()
            if resultado is None:
                if x == 0:
                    print("Nada encontrado.")
                break
            print(f"Nome: {resultado[0]}\nTelefone: {resultado[1]}")
            x += 1
```

Agora utilizamos um parâmetro, como fizemos antes para inserir nossos registros. Um detalhe importante é que escrevemos `(nome,)`, repare a vírgula após `nome`. Esse detalhe é importante, pois o segundo parâmetro do método `execute` é uma tupla, e, em Python, tuplas com apenas um elemento são escritas com uma vírgula após o primeiro valor. Veja também que utilizamos a variável `x` para contar quantos resultados obtivemos. Como o método `fetchone` retorna **`None`** quando todos os registros foram recebidos, verificamos se `x == 0`, para saber se algo já havia sido obtido anteriormente ou se devemos imprimir uma mensagem dizendo que nada foi encontrado. Em Python, `x == 0` produz o mesmo resultado de apenas `x`. Dessa forma, você poderia ter escrito a linha como **`if`** `x:` em vez de **`if`** `x == 0`. Isso acontece, pois valores diferentes de 0 no Python são considerados como verdadeiros e **`None`**, 0, "", {}, [] como falsos.

**Exercício 11.3** Escreva um programa que realize consultas do banco de dados preços.db, criado no Exercício 11.1. O programa deve perguntar o nome do produto e listar seu preço.

**Exercício 11.4** Modifique o programa do Exercício 11.3 de forma a perguntar dois valores e listar todos os produtos com preços entre esses dois valores.

## 11.5 Atualizando registros

Já sabemos como criar tabelas, inserir registros e fazer consultas simples. Vamos começar a usar o comando `update` para alterar nossos registros. Por exemplo, vamos alterar o registro com o telefone de "Nilo" para "12345-6789":

```
update agenda set telefone = "12345-6789" where nome = 'Nilo'
```

A cláusula `where` funciona como no comando `select`, ou seja, ela avalia uma expressão lógica que, quando verdadeira, inclui o registro na lista de registros a modificar. A segunda parte do comando `update` é a cláusula `set`. Essa cláusula é usada para indicar o que fazer nos registros selecionados pela expressão do `where`. No exemplo, `set telefone = "12345-6789"` muda o conteúdo do campo `telefone` para "12345-6789". O comando inteiro poderia ser lido como: atualize os registros da tabela agenda, alterando o telefone para "12345-6789" em todos os registros em que o campo `nome` é igual a "Nilo". Vejamos o programa:

```
import sqlite3
from contextlib import closing
```

```
with sqlite3.connect("agenda.db") as conexão:
    with closing(conexão.cursor()) as cursor:
        cursor.execute("""update agenda
                          set telefone = '12345-6789'
                          where nome = 'Nilo'""")
    conexão.commit()
```

Nesse exemplo, utilizamos constantes, logo não precisamos usar parâmetros. As mesmas regras que aprendemos para o comando `select` se aplicam ao comando `update`. Se os valores não forem constantes, você tem de utilizar parâmetros.

O comando `update` pode alterar mais de um registro de uma só vez. Faça uma cópia do arquivo `agenda.db` e experimente modificar o programa anterior, retirando a cláusula `where`:

```
update agenda set telefone = "12345-6789"
```

Você verá que todos os registros foram modificados:

```
Nome: Nilo
Telefone: 12345-6789
Nome: João
Telefone: 12345-6789
Nome: André
Telefone: 12345-6789
Nome: Maria
Telefone: 12345-6789
```

Sem a cláusula `where`, todos os registros serão selecionados e alterados. Vamos utilizar a propriedade `rowcount` de nosso cursor para saber quantos registros foram alterados por nosso `update`. Veja o programa com essas alterações:

```
# Programa 11.6 - Exemplo de update sem where e com rowcount
import sqlite3
from contextlib import closing

with sqlite3.connect("agenda.db") as conexão:
    with closing(conexão.cursor()) as cursor:
        cursor.execute("""update agenda
                          set telefone = '12345-6789' """)
        print("Registros alterados: ", cursor.rowcount)
```

Não se esqueça de que, após modificar o banco de dados, precisamos chamar o método `commit`, como fizemos ao inserir os registros. Caso nos esqueçamos, as alterações serão perdidas.

A propriedade rowcount é muito interessante para confirmarmos o resultado de comandos de atualização, como update. Essa propriedade não funciona com select, retornando sempre -1. Por isso, no Programa 11.5, contamos os registros retornados por nosso select em vez de usarmos rowcount. No caso de update, poderíamos fazer uma verificação de quantos registros seriam alterados antes de chamarmos o commit, como:

```
import sqlite3
from contextlib import closing

with sqlite3.connect("agenda.db") as conexão:
    with closing(conexão.cursor()) as cursor:
        cursor.execute("""update agenda
                          set telefone = '12345-6789' """)
        print("Registros alterados: ", cursor.rowcount)
        if cursor.rowcount == 1:
            conexão.commit()
            print("Alterações gravadas")
        else:
            conexão.rollback()
            print("Alterações abortadas")
```

Utilizamos o valor de rowcount para decidir se as alterações deveriam ser registradas ou ignoradas. Como já sabemos, o método commit grava as alterações. O método rollback faz o inverso, abortando as alterações e deixando o banco de dados como antes. Os métodos commit e rollback fazem o controle de transações do banco de dados. Podemos entender uma transação como um conjunto de operações que deve ser executado completamente. Isso significa operações que não fazem sentido, salvo se realizadas em um só grupo. Se a execução do grupo falhar, todas as alterações causadas durante a transação corrente devem ser revertidas (rollback). Caso tudo ocorra como planejado, as operações serão armazenadas definitivamente no banco de dados (commit). Veremos outros exemplos mais adiante.

**Exercício 11.5** Escreva um programa que aumente o preço de todos os produtos do banco *preços.db* em 10%.

**Exercício 11.6** Escreva um programa que pergunte o nome do produto e um novo preço. Usando o banco *preços.db*, atualize o preço desse produto no banco de dados.

## 11.6 Apagando registros

Além de inserir, consultar e alterar registros, podemos também apagá-los. O comando `delete` apaga registros com base em um critério de seleção, especificado na cláusula `where` que já conhecemos. Faça outra cópia do arquivo `agenda.db`. Copie o antigo banco de dados, com os registros antes de executarmos o Programa 11.6.

A sintaxe do comando `delete` é:

```
delete from agenda where nome = 'Maria'
```

Ou seja, apague da tabela `agenda` todos os registros com nome igual a "Maria":

```
import sqlite3
from contextlib import closing

with sqlite3.connect("agenda.db") as conexão:
    with closing(conexão.cursor()) as cursor:
        cursor.execute("""delete from agenda
                                where nome = 'Maria' """)
        print("Registros apagados: ", cursor.rowcount)
        if cursor.rowcount == 1:
            conexão.commit()
            print("Alterações gravadas")
        else:
            conexão.rollback()
            print("Alterações abortadas")
```

Utilizamos o método `rowcount` para ter certeza de que estávamos apagando apenas um registro. Assim como no comando `insert` e `update`, você precisa chamar `commit` para gravar as alterações ou `rollback`, caso contrário.

## 11.7 Simplificando o acesso sem cursores

A interface de banco de dados do Python nos permite executar alguns comandos utilizando diretamente o objeto da conexão, sem criarmos explicitamente um cursor. Vejamos o programa seguinte, que é uma versão simplificada do Programa 11.1.

```
import sqlite3
with sqlite3.connect("agenda.db") as conexão:
    for registro in conexão.execute("select * from agenda"): ❶
        print(f"Nome: {registro[0]}\nTelefone: {registro[1]}" % (registro))
```

Utilizamos a estrutura **with** para facilitar o fechamento da conexão. Em ❶, `conexão.execute` retorna um cursor que pode ser usado com **for**. Você pode também utilizar o método `executemany` diretamente com o objeto conexão. Essa utilização simplificada funciona muito bem com SQLite, mas não faz parte da interface padrão de banco de dados do Python, a DB-API 2.0. Ao utilizar cursores, você obedece a DB-API 2.0 que é implementada por outros bancos de dados, simplificando a migração de seu código para outros bancos de dados, como o MySQL ou MariaDB.

## 11.8 Acessando os campos como em um dicionário

Acessar os campos por posição nem sempre é tão fácil. Em Python, usando SQLite, podemos acessá-los pelo nome, adicionando uma linha:

```
conexão.row_factory = sqlite3.Row
```

Vejamos o programa completo:

```
import sqlite3
from contextlib import closing

with sqlite3.connect("agenda.db") as conexão:
    conexão.row_factory = sqlite3.Row
    with closing(conexão.cursor()) as cursor:
        for registro in cursor.execute("select * from agenda"):
            print(f"Nome: {registro['nome']}\nTelefone: {registro['telefone']}")
```

Dessa forma, registro pode ser acessado como se fosse um dicionário, no qual o nome do campo é usado como chave. Outra facilidade que essa linha traz é que as chaves são aceitas independentemente se escrevermos o nome dos campos em maiúsculas ou minúsculas. Por exemplo:

```
f"Nome: {registro['Nome']}\nTelefone: {registro['Telefone']}"
```

## 11.9 Gerando uma chave primária

Até agora, trabalhamos apenas com campos normais, ou seja, campos que contêm dados. Conforme nossas tabelas crescem, trabalhar com os dados pode não ser a melhor solução, e precisaremos acrescentar campos para manter o banco de dados. Uma dessas necessidades é identificar cada registro de maneira única. Nós podemos utilizar dados que não se repetem, ou que não deveriam se repetir, como o nome da pessoa, como uma chave primária. Podemos entender uma chave

primária como a chave de um dicionário, mas, nesse caso, para tabelas em nosso banco de dados. Qualquer campo ou um conjunto de campos podem servir de chave primária. Uma alternativa oferecida pelo SQLite é a geração automática de chaves. Nesse caso, o banco se encarrega de criar números únicos para cada registro.

Vamos implementar outro banco de dados, com a população de cada estado do Brasil. Veremos como deixar o SQLite gerar uma chave primária automaticamente:

```
create table estados(id integer primary key autoincrement,
                     nome text, população integer)
```

Ao criarmos a tabela estados, estamos especificando três campos: `id`, `nome` e `população`. Veja que `id` e `população` são do tipo `integer`, ou seja, números inteiros (**`int`**). `id` é o campo que escolhemos para ser a chave primária dessa tabela, e escrevemos `primary key autoincrement` para que o SQLite gere esses números automaticamente. Entenda `id` como a abreviação de identificador único ou identidade. *Primary key* significa chave primária.

O programa a seguir cria o banco de dados `brasil.db`, a tabela estados e também inclui o nome e a população de todos os estados brasileiros. Os dados foram extraídos da Wikipédia (`http://pt.wikipedia.org/wiki/Anexo:Lista_de_unidades_federativas_do_Brasil_por_popula%C3%A7%C3%A3o`).

```
import sqlite3
from contextlib import closing
dados = [["São Paulo", 43663672], ["Minas Gerais", 20593366], ["Rio de Janeiro",
   16369178], ["Bahia", 15044127], ["Rio Grande do Sul", 11164050], ["Paraná",
   10997462], ["Pernambuco", 9208511], ["Ceará", 8778575], ["Pará", 7969655],
   ["Maranhão", 6794298], ["Santa Catarina", 6634250], ["Goiás", 6434052],
   ["Paraíba", 3914418], ["Espírito Santo", 3838363], ["Amazonas", 3807923],
   ["Rio Grande do Norte", 3373960], ["Alagoas", 3300938], ["Piauí", 3184165],
   ["Mato Grosso", 3182114], ["Distrito Federal", 2789761], ["Mato Grosso do Sul",
   2587267], ["Sergipe", 2195662], ["Rondônia", 1728214], ["Tocantins", 1478163],
   ["Acre", 776463], ["Amapá", 734995], ["Roraima", 488072]]
with sqlite3.connect("brasil.db") as conexão:
    conexão.row_factory = sqlite3.Row
    with closing(conexão.cursor()) as cursor:
        cursor.execute("""create table estados(
                          id integer primary key autoincrement,
                          nome text,
                          população integer
                      )""")
        cursor.executemany("insert into estados(nome, população) values(?,?)", dados)
    conexão.commit()
```

O valor do campo id será gerado automaticamente. Uma vez que temos a população dos estados, vamos fazer uma consulta para listar os estados em ordem alfabética. Ao executar o próximo programa, observe os valores gerados no campo id, no caso, valores numéricos de 1 a 27.

```
import sqlite3
from contextlib import closing

with sqlite3.connect("brasil.db") as conexão:
    conexão.row_factory = sqlite3.Row
    print(f"{'Id':3s} {'Estado':-20s} {'População':12s}")
    print("=" * 37)
    for estado in conexão.execute("select * from estados order by nome"):
        print(f"{estado['id']:3d} {estado['nome']:-20s} {estado['população']:12d}")
```

A grande diferença é que estamos utilizando a cláusula order by para ordenar os resultados de nossa consulta; nesse caso, pelo campo nome.

```
select * from estados order by nome
```

Modifique o programa para que os estados sejam impressos pela população, usando a consulta:

```
select * from estados order by população
```

Execute o programa novamente e veja que os estados foram agora impressos pela população, mas da menor para a maior. Embora essa seja a ordem normal, quando trabalhamos com lista de estados por população, esperamos ver do estado mais populoso para o menos populoso, ou seja, na ordem inversa (decrescente) dos valores. Vejamos esse resultado ao adicionarmos desc após o nome do campo:

```
select * from estados order by população desc
```

## 11.10 Alterando a tabela

Vamos acrescentar mais alguns campos a nossa tabela de estados. Um campo para a região do Brasil e outro para a sigla do estado. Em SQL, o comando utilizado para alterar os campos de uma tabela é o alter table.

```
alter table estados add sigla text
alter table estados add região text
```

O comando `alter table` do SQLite é limitado se comparado com outros bancos de dados. Em outros bancos, pode-se alterar vários campos com um só `alter table`, mas, no SQLite, somos obrigados a alterar um campo de cada vez. As limitações do `alter table` do SQLite não param por aí. Por isso, planeje suas tabelas com cuidado e, caso precise realizar grandes mudanças, prefira criar uma outra tabela com as alterações e copiar os dados da tabela antiga. Execute o programa para alterar a tabela estados e adicionar os campos sigla e região:

```
import sqlite3

with sqlite3.connect("brasil.db") as conexão:
    conexão.execute("""alter table estados
                       add sigla text""")
    conexão.execute("""alter table estados
                       add região text""")
```

Agora que a tabela possui os novos campos, vamos alterar nossos registros e preencher a região e sigla de cada estado. Execute o programa:

```
import sqlite3
dados = [["SP", "SE", "São Paulo"], ["MG", "SE", "Minas Gerais"], ["RJ", "SE", "Rio
   de Janeiro"], ["BA", "NE", "Bahia"], ["RS", "S", "Rio Grande do Sul"], ["PR",
   "S", "Paraná"], ["PE", "NE", "Pernambuco"], ["CE", "NE", "Ceará"], ["PA", "N",
   "Pará"], ["MA", "NE", "Maranhão"], ["SC", "S", "Santa Catarina"], ["GO", "CO",
   "Goiás"], ["PB", "NE", "Paraíba"], ["ES", "SE", "Espírito Santo"], ["AM", "N",
   "Amazonas"], ["RN", "NE", "Rio Grande do Norte"], ["AL", "NE", "Alagoas"], ["PI",
   "NE", "Piauí"], ["MT", "CO", "Mato Grosso"], ["DF", "CO", "Distrito Federal"],
   ["MS", "CO", "Mato Grosso do Sul"], ["SE", "NE", "Sergipe"], ["RO", "N",
   "Rondônia"], ["TO", "N", "Tocantins"], ["AC", "N", "Acre"], ["AP", "N", "Amapá"],
   ["RR", "N", "Roraima"]]

with sqlite3.connect("brasil.db") as conexão:
    conexão.executemany("""update estados
                           set sigla = ?,
                           região = ?
                           where nome = ?""", dados)
```

Agora nosso banco de dados possui uma tabela estados com a população, sigla e região de cada estado. Esses novos campos permitirão utilizarmos funções de agregação da linguagem SQL: `count`, `min`, `max`, `avg` e `sum`.

## 11.11 Agrupando dados

Um banco de dados pode realizar operações de agrupamento de dados facilmente. Podemos, por exemplo, solicitar o valor mínimo de um grupo de registros, assim como também o máximo ou a média desses valores. No entanto, temos de modificar nossos comandos SQL para indicar uma cláusula de agrupamento, ou seja, devemos indicar como o banco de dados deve agrupar nossos registros.

Vejamos como realizar um grupo simples e exibir quantos registros fazem parte desse grupo, usando a função `count`. A cláusula SQL que indica agrupamento é `group by`, seguida do nome dos campos que compõem o grupo. Imagine que o banco vai concatenar cada um desses campos, criando um valor para cada registro. Vamos chamar esse valor de "chave de grupo". Todos os registros com a mesma chave de grupo fazem parte do mesmo grupo e serão representados por apenas um registro na consulta de seleção. Essa consulta com grupo só pode conter os campos utilizados para compor a chave do grupo e funções de agrupamento de dados, como `min` (mínimo), `max` (máximo), `avg` (média), `sum` (soma) e `count` (contagem).

Um exemplo concreto como nosso banco de dados é agrupar os estados por região. A consulta seria algo como:

```
select região, count(*) from estados group by região
```

Esse comando utiliza a cláusula `group by` região para especificar a chave de grupo. Dessa forma, todos os registros que pertencem à mesma região são agrupados. Observe que os campos após o `select` incluem região e `count(*)`. O campo região pôde ser incluído, pois faz parte da chave de grupo especificada na `group by`. A função `count(*)` retorna quantos registros fazem parte do grupo. Vejamos o resultado do programa:

```
import sqlite3
print("Região Número de Estados")
print("====== ================")
with sqlite3.connect("brasil.db") as conexão:
    for região in conexão.execute("""
        select região, count(*)
        from estados
        group by região"""):
        print("{0:6} {1:17}".format(*região))
```

Resultado da execução do programa:

```
Região Número de Estados
====== =================
CO                     4
N                      7
NE                     9
S                      3
SE                     4
```

Vamos adicionar as funções `min`, `max`, `sum`, `avg` no campo população. Veja o novo programa:

```
import sqlite3

print("Região Estados População  Mínima    Máxima      Média    Total (soma)")
print("====== =======          ========= ========== ==========  ============")
with sqlite3.connect("brasil.db") as conexão:
    for região in conexão.execute("""
        select região, count(*), min(população),
               max(população), avg(população), sum(população)
        from estados
        group by região"""):
        print("{0:6} {1:7} {2:18,} {3:10,} {4:10,.0f} {5:13,}".format(*região))
    print("\nBrasil: {0:6} {1:18,} {2:10,} {3:10,.0f} {4:13,}".format(
        *conexão.execute("""
            select count(*), min(população), max(população),
                   avg(população), sum(população) from estados""").fetchone()))
```

Resultado do programa:

```
Região Estados População  Mínima    Máxima      Média    Total (soma)
====== =======          ========= ========== ==========  ============
CO           4          2,587,267  6,434,052  3,748,298    14,993,194
N            7            488,072  7,969,655  2,426,212    16,983,485
NE           9          2,195,662 15,044,127  6,199,406    55,794,654
S            3          6,634,250 11,164,050  9,598,587    28,795,762
SE           4          3,838,363 43,663,672 21,116,145    84,464,579
Brasil:     27            488,072 43,663,672  7,445,618   201,031,674
```

Conseguimos calcular a população mínima, máxima, média e total de cada região e também para o Brasil. Veja que na segunda consulta, a que calcula os dados para o Brasil, não utilizamos a cláusula `group by`, fazendo com que todos os registros façam parte do grupo.

Ao utilizarmos as funções de agregação e a cláusula `group by`, podemos continuar usando tudo que já aprendemos em SQL, como as cláusulas `where` e `order by`. Vejamos o mesmo programa, mas com as linhas ordenadas pela população total de cada região em ordem decrescente:

```
import sqlite3
print("Região Estados População  Mínima    Máxima      Média    Total (soma)")
print("====== =======          ========= ========== ==========  ============")
with sqlite3.connect("brasil.db") as conexão:
    for região in conexão.execute("""
        select região, count(*), min(população),
               max(população), avg(população), sum(população)
        from estados
        group by região
        order by sum(população) desc"""):
        print("{0:6} {1:7} {2:18,} {3:10,} {4:10,.0f} {5:13,}".format(*região))
    print("\nBrasil: {0:6} {1:18,} {2:10,} {3:10,.0f} {4:13,}".format(
        *conexão.execute("""
            select count(*), min(população), max(população),
                   avg(população), sum(população) from estados""").fetchone()))
```

Apenas acrescentamos a linha `order by sum(população) desc` no final de nossa consulta. Veja que repetimos a função de agregação `sum(população)` para indicar que a ordenação será feita pela soma da população. Você pode utilizar a cláusula **as** do SQL para dar nomes às colunas de uma consulta. Veja a consulta modificada para usar **as** e criar uma coluna `tpop` para a soma da população:

```
select região, count(*), min(população), max(população), avg(população),
          sum(população) as tpop
   from estados group by região
   order by tpop desc
```

Veja que escrevemos `sum(população) as tpop`, dando o nome `tpop` à soma. Depois, utilizamos o nome `tpop` na cláusula do `order by`. Esse tipo de construção evita a repetição da função na consulta e facilita a leitura.

Resultado da execução do programa:

```
Região Estados População  Mínima    Máxima      Média    Total (soma)
====== ======= ========= ========== ========== ==========  ============
SE           4         3,838,363 43,663,672 21,116,145    84,464,579
NE           9         2,195,662 15,044,127  6,199,406    55,794,654
S            3         6,634,250 11,164,050  9,598,587    28,795,762
```

```
N          7          488,072  7,969,655  2,426,212    16,983,485
CO         4        2,587,267  6,434,052  3,748,298    14,993,194

Brasil:    27         488,072 43,663,672  7,445,618   201,031,674
```

Podemos também filtrar os resultados após o agrupamento, usando a cláusula having. Para entender a diferença entre where e having, imagine que where é executada antes do agrupamento, selecionando os registros que farão parte do resultado, antes de o agrupamento ser realizado. A cláusula having avalia o resultado do agrupamento e decide quais farão parte do resultado final. Por exemplo, podemos escolher apenas as regiões com mais de 5 estados. Como a quantidade de estados por região só é conhecida após o agrupamento (group by), essa condição deve aparecer em uma cláusula having.

```
select região, count(*), min(população),
          max(população), avg(população), sum(população) as tpop
          from estados group by região
       having count(*) > 5
          order by tpop desc
```

Veja o programa completo utilizando having:

```
import sqlite3
print("Região Estados População  Mínima    Máxima      Média    Total (soma)")
print("====== =======          ========= ========== ==========  ============")
with sqlite3.connect("brasil.db") as conexão:
    for região in conexão.execute("""
        select região, count(*), min(população),
               max(população), avg(população), sum(população) as tpop
        from estados
        group by região
        having count(*) > 5
        order by tpop desc"""):
        print("{0:6} {1:7} {2:18,} {3:10,} {4:10,.0f} {5:13,}".format(*região))
```

Resultando em:

```
Região Estados População  Mínima    Máxima      Média    Total (soma)
====== =======          ========= ========== ==========  ============
NE          9          2,195,662 15,044,127  6,199,406    55,794,654
N           7            488,072  7,969,655  2,426,212    16,983,485
```

Uma vez que apenas as regiões Norte (N) e Nordeste (NE) possuem mais de 5 estados.

## 11.12 Trabalhando com datas

Embora o SQLite trabalhe com datas, o tipo DATE não é suportado diretamente, gerando uma certa confusão entre datas e strings. Vamos criar uma tabela com um campo do tipo data.

```
import sqlite3
feriados = [["2018-01-01", "Confraternização Universal"], ["2018-04-21",
   "Tiradentes"], ["2018-05-01", "Dia do trabalhador"], ["2018-09-07",
   "Independência"], ["2018-10-12", "Padroeira do Brasil"], ["2018-11-02",
   "Finados"], ["2018-11-15", "Proclamação da República"], ["2018-12-25", "Natal"]]
with sqlite3.connect("brasil.db") as conexão:
    conexão.execute("create table feriados(id integer primary key autoincrement,
   data date, descrição text)")
    conexão.executemany("insert into feriados(data,descrição) values (?,?)",
   feriados)
```

Criamos a tabela feriados e inserimos algumas datas. Observe que escrevemos as datas no formato ISO 8601 (http://pt.wikipedia.org/wiki/ISO_8601), ou seja: ANO-MÊS-DIA. Nesse formato, a data do Natal (25/12/2018) é escrita como 2018-12-25. Escrever as datas nesse formato é uma característica do SQLite. Sempre escreva suas datas no formato ISO ao trabalhar com esse gerenciador de banco de dados. Observe que utilizamos o tipo `date` (data) na coluna `data`. Modifique o ano de 2018 para o ano corrente, caso necessário.

Vejamos como acessar esses valores no programa:

```
import sqlite3
with sqlite3.connect("brasil.db") as conexão:
    for feriado in conexão.execute("select * from feriados"):
        print(feriado)
```

Que resulta em:

```
(1, '2018-01-01', 'Confraternização Universal')
(2, '2018-04-21', 'Tiradentes')
(3, '2018-05-01', 'Dia do trabalhador')
(4, '2018-09-07', 'Independência')
(5, '2018-10-12', 'Padroeira do Brasil')
(6, '2018-11-02', 'Finados')
(7, '2018-11-15', 'Proclamação da República')
(8, '2018-12-25', 'Natal')
```

Acessamos o campo data como fazemos até então, sem algum procedimento especial. Veja que, ao imprimirmos a tupla `feriado` com o resultado de nossa

seleção, o campo `data` foi impresso como uma string qualquer. Nada impede que utilizemos strings para representar datas, como fizemos ao criar o banco de dados, mas campos datas são mais interessantes, pois podemos facilmente consultar o dia da semana e também realizar operações com datas.

Vamos modificar nossa conexão com o SQLite de forma a solicitar o processamento dos tipos de campo em nossas consultas. Ao solicitarmos a conexão, devemos passar `detect_types=sqlite3.PARSE_DECLTYPES` como parâmetro. Vejamos o novo programa com essa modificação:

```
import sqlite3

with sqlite3.connect("brasil.db", detect_types=sqlite3.PARSE_DECLTYPES) as conexão:
    for feriado in conexão.execute("select * from feriados"):
        print(feriado)
```

O resultado é bem diferente:

```
(1, datetime.date(2018, 1, 1), 'Confraternização Universal')
(2, datetime.date(2018, 4, 21), 'Tiradentes')
(3, datetime.date(2018, 5, 1), 'Dia do trabalhador')
(4, datetime.date(2018, 9, 7), 'Independência')
(5, datetime.date(2018, 10, 12), 'Padroeira do Brasil')
(6, datetime.date(2018, 11, 2), 'Finados')
(7, datetime.date(2018, 11, 15), 'Proclamação da República')
(8, datetime.date(2018, 12, 25), 'Natal')
```

Veja que os valores do campo `data` agora são objetos da classe `datetime.date`. Isso evita termos de fazer a conversão manualmente de string para `datetime.date`.

Agora vamos utilizar o método `strftime` do objeto da classe `datetime.date` para exibir apenas o dia e o mês da data, sem o ano:

```
import sqlite3
with sqlite3.connect("brasil.db", detect_types=sqlite3.PARSE_DECLTYPES) as conexão:
    conexão.row_factory = sqlite3.Row
    for feriado in conexão.execute("select * from feriados"):
        print(f"{feriado['data'].strftime(%d/%m)} {feriado['descrição']}")
```

Voltamos a utilizar `row_factory` para acessarmos os campos por nome, como em um dicionário. O método `strftime` foi utilizado com a máscara `"%d/%m"` para exibir apenas o dia e o mês. Você pode verificar os formatos aceitos por `strftime` na Tabela 9.3.

Vejamos o resultado do programa:

```
01/01 Confraternização Universal
21/04 Tiradentes
01/05 Dia do trabalhador
07/09 Independência
12/10 Padroeira do Brasil
02/11 Finados
15/11 Proclamação da República
25/12 Natal
```

Vejamos um pouco o que podemos fazer com os objetos do módulo `datetime`.

```
import sqlite3
import datetime
hoje = datetime.date.today()
hoje60dias = hoje + datetime.timedelta(days=60)
with sqlite3.connect("brasil.db", detect_types=sqlite3.PARSE_DECLTYPES) as conexão:
    conexão.row_factory = sqlite3.Row
    for feriado in conexão.execute("select * from feriados where data >= ? and data
    <= ?", (hoje, hoje60dias)):
        print(f"{feriado['data']:%d/%m} {feriado['descrição']}")
```

O programa utiliza objetos do módulo `datetime`. Em `hoje`, guardamos a data atual (`datetime.date.today()`). Em `hoje60dias`, utilizamos um objeto do tipo `datetime.timedelta` para acrescentar 60 dias à data atual. Com esses dois objetos `date`, podemos utilizar a cláusula `where` do SQLite para selecionar os feriados entre `hoje` e `hoje60dias`. Consulte a documentação do Python para saber mais sobre o módulo `datetime`. Os objetos das classes `timedelta` e `datetime.datetime` são bastante úteis, caso você precise realizar operações com datas e guardar a hora com a data (`datetime`).

## 11.13 Chaves e relações

Agora que já sabemos o básico de como manipular registros em nosso banco de dados, veremos conceitos mais avançados que permitirão trabalharmos com várias tabelas. Para selecionarmos nossos registros, vimos que precisamos construir expressões lógicas que identifiquem ou que permitam a seleção desses registros. No caso de nossa agenda, o campo `nome` foi usado em nossas expressões, mas utilizar dados como critério de seleção não é uma boa ideia a longo termo. Dados podem mudar e se repetir entre várias tabelas. Por exemplo, imagine a situação em que nossa agenda teria vários telefones por pessoa, como na Tabela 11.2:

*Tabela 11.2 – Agenda com uma tabela*

| nome | número | tipo |
|---|---|---|
| Nilo | 12345-6789 | Casa |
| Nilo | 98745-4321 | Celular |

Nesse exemplo, poderíamos simplesmente adicionar dois registros com o mesmo nome, mas estaríamos complicando nosso trabalho mais tarde, pois, se quiséssemos mudar o telefone de um dos registros, não poderíamos utilizar `nome = "Nilo"` como critério de seleção, seríamos obrigados a utilizar uma condição composta de `nome` e `telefone`.

Esse problema poderia ser resolvido adicionando-se uma chave que identificasse cada pessoa de forma única. Essa chave pode ser um simples número, desde que único, vamos chamá-la de identificador, ou simplesmente `id`. A Tabela 11.3 mostra nossos dados com esse novo campo.

*Tabela 11.3 – Agenda com uma tabela e uma chave*

| id | nome | número | tipo |
|---|---|---|---|
| 1 | Nilo | 12345-6789 | Casa |
| 1 | Nilo | 98745-4321 | Celular |

Embora nossos dados estejam em melhor forma, ainda estamos repetindo o campo `nome` várias vezes. E o campo `id` não identifica unicamente cada registro. Esse tipo de problema é chamado de redundância de dados. Em uma base de dados, quanto menos redundância tivermos em nossos dados, mais fácil será de mantê-los. Por exemplo, imagine que queiramos mudar o nome Nilo para adicionar também Menezes. Teríamos de atualizar os dois registros, pois guardamos a mesma informação em dois lugares (registros) diferentes.

Uma melhor forma de representar esses dados é dividindo nossos dados em várias tabelas. Por exemplo, uma tabela para nome e outra para telefone. Vejamos as tabelas 11.4 e 11.5 com essa nova divisão. Veja que o campo `id` na Tabela 11.4 pode ser usado como chave primária. A chave primária (`id`) da tabela nomes foi copiada na Tabela 11.5, no campo `id_nome`.

*Tabela 11.4 – Tabela nomes*

| id | Nome |
|---|---|
| 1 | Nilo |

*Tabela 11.5 – Tabela Telefones*

| id_nome | número | tipo |
|---|---|---|
| 1 | 12345-6789 | Casa |
| 1 | 98745-4321 | Celular |

Dessa forma, armazenamos o `nome` em apenas um lugar. Ainda temos outros problemas, pois agora nossos telefones não possuem um identificador único. Veja como ficaria nossa tabela, acrescentando-se uma chave a Telefones, Tabela 11.6:

*Tabela 11.6 – Tabela telefones*

| id | id_nome | número | tipo |
|---|---|---|---|
| 1 | 1 | 12345-6789 | Casa |
| 2 | 1 | 98745-4321 | Celular |

Assim, uma chave primária é um campo de um registro que o identifica de forma única na tabela. Resolvemos nosso problema de redundância, mas como acessar esses dados em tabelas diferentes com comandos SQL? Bem, vamos utilizar o comando `select`, mas com várias tabelas e especificar uma forma de as interligar.

```
select * from nomes, telefones where nomes.id = telefones.id_nomes
```

Esse comando difere de nossos outros exemplos por utilizar mais de uma tabela, após a cláusula `from`. Uma vez que utilizamos várias tabelas, somos obrigados a especificar como essas tabelas se relacionam; caso contrário, obteremos o que é chamado de produto cartesiano, em que nosso resultado conterá a combinação de cada registro da primeira tabela, com cada registro da segunda. É esse relacionamento que é especificado em `nomes.id = telefones.id_nomes`, ou seja, especificamos um critério que liga as duas tabelas; no caso, quando o campo `id` da tabela nomes (`nomes.id`) for igual ao campo `id_nomes` da tabela telefones (`telefones.id_nomes`).

Mas o tipo do telefone ainda se repete, o que pode levar a resultados indesejáveis em nossa agenda. Vamos criar outra tabela para guardar os tipos de telefone de forma a não os repetir. Veja o resultado nas tabelas 11.7 e 11.8.

*Tabela 11.7 Tabela telefones com o campo id_tipo*

| id | id_nome | número | id_tipo |
|---|---|---|---|
| 1 | 1 | 12345-6789 | 1 |
| 2 | 1 | 98745-4321 | 2 |

*Tabela 11.8 Tabela Tipos*

| id | descrição |
|---|---|
| 1 | Casa |
| 2 | Celular |

Vejamos os comandos SQL para criar essas tabelas:

```
create table tipos(id integer primary key autoincrement,
                   descrição text);
create table nomes(id integer primary key autoincrement,
                   nome text);
create table telefones(id integer primary key autoincrement,
                       id_nome integer, número text,
                       id_tipo integer);
```

Com o uso de `primary key autoincrement`, como já vimos anteriormente, o SQLite se encarregará de gerar os números que utilizaremos em nossas chaves primárias. Agora, podemos revistar a agenda do Capítulo 10 e convertê-la para utilizar um banco de dados em vez de um simples arquivo-texto.

## 11.14 Convertendo a agenda para utilizar um banco de dados

Converter a agenda para um banco de dados nos levará a enfrentar um problema de mapeamento entre objetos e os bancos de dados relacionais, como o SQLite. Um dos maiores problemas nesse tipo de mapeamento é manter os dados entre nosso programa e o banco de dados sincronizados. Existem bibliotecas inteiras escritas apenas para resolver esse tipo de problema, usando o que é chamado de Mapeamento Objeto Relacional (*Object-Relational Mapping*, ORM). Em nossa agenda, temos uma lista de registros, cada um com um nome e uma lista de telefones. Cada telefone possui um tipo pré-cadastrado.

Primeiro, vamos criar uma subclasse de `ListaÚnica` para controlar os registros apagados de nossas listas. Depois, criaremos métodos em uma outra classe, chamada `DBAgenda`, responsável por manter o banco de dados e executar as operações da agenda. Uma mudança nessa nova versão da agenda é que carregamos o registro apenas quando precisamos carregar. Ao voltarmos ao menu principal, todas as mudanças já estarão salvas no banco de dados, fazendo as opções Lê e Grava inúteis.

```
class DBListaÚnica(ListaÚnica):
    def __init__(self, elem_class):
        super().__init__(elem_class)
        self.apagados = []
    def remove(self, elem):
        if elem.id is not None:
            self.apagados.append(elem.id)
        super().remove(elem)
    def limpa(self):
        self.apagados = []
class DBNome(Nome):
    def __init__(self, nome, id_=None):
        super().__init__(nome)
        self.id = id_
class DBTipoTelefone(TipoTelefone):
    def __init__(self, id_, tipo):
        super().__init__(tipo)
        self.id = id_
class DBTelefone(Telefone):
    def __init__(self, número, tipo=None, id_=None, id_nome=None):
        super().__init__(número, tipo)
        self.id = id_
        self.id_nome = id_nome
class DBTelefones(DBListaÚnica):
    def __init__(self):
        super().__init__(DBTelefone)
class DBTiposTelefone(ListaÚnica):
    def __init__(self):
        super().__init__(DBTipoTelefone)
class DBDadoAgenda:
    def __init__(self, nome):
        self.nome = nome
        self.telefones = DBTelefones()
    @property
    def nome(self):
        return self.__nome
    @nome.setter
    def nome(self, valor):
        if not isinstance(type(valor), DBNome):
            raise TypeError("nome deve ser uma instância da classe DBNome")
        self.__nome = valor
```

```
    def pesquisaTelefone(self, telefone):
        posição = self.telefones.pesquisa(DBTelefone(telefone))
        if posição == -1:
            return None
        else:
            return self.telefones[posição]
```

A classe DBListaÚnica herda de nossa classe ListaÚnica e sua principal função é manter uma lista de id apagados. Isso nos permitirá apagar os elementos de nossas listas e, numa fase seguinte, apagá-los do banco de dados. A classe DBListaÚnica só pode trabalhar com classes que possuam um atributo id.

As classes DBNome e DBTelefone derivam de Nome e Telefone respectivamente. A principal diferença é que agora elas incluem o atributo id. Veja que esse atributo é um parâmetro opcional, pois, ao criarmos nossos objetos, eles não terão ainda suas chaves primárias. Além disso, usaremos o fato de que objetos sem id provavelmente acabaram de ser criados e precisam ser inseridos no banco de dados. Isso ficará mais claro no programa completo.

Já na classe DBDadosAgenda modificamos o tipo da lista de telefones de Telefones para DBTelefones. A classe DBTelefones é uma derivação de DBListaÚnica que aceita apenas elementos do tipo DBTelefone. Fizemos o mesmo entre DBTipoTelefone e DBTiposTelefones.

Até agora, fizemos apenas a mudança dos tipos, em preparação para trabalhar com o banco de dados. A principal mudança foi o novo atributo id que acrescentamos em todas as nossas classes. É o valor desse campo que utilizaremos em nossas consultas, alterações e remoções.

O programa a seguir apresenta a classe DBAgenda, que substituirá a classe Agenda. A classe DBAgenda mantém o banco de dados em sincronia com as classes e objetos em memória, sendo responsável por todas as operações com o banco. Esse tipo de construção em camadas evita termos de escrever o código de manipulação e criação do banco na classe AppAgenda.

```
BANCO = """
create table tipos(id integer primary key autoincrement,
                   descrição text);
create table nomes(id integer primary key autoincrement,
                   nome text);
create table telefones(id integer primary key autoincrement,
                   id_nome integer,
                   número text,
                   id_tipo integer);
```

```
insert into tipos(descrição) values ("Celular");
insert into tipos(descrição) values ("Fixo");
insert into tipos(descrição) values ("Fax");
insert into tipos(descrição) values ("Casa");
insert into tipos(descrição) values ("Trabalho");
"""

class DBAgenda:
    def __init__(self, banco):
        self.tiposTelefone = DBTiposTelefone()
        self.banco = banco
        novo = not os.path.isfile(banco)
        self.conexão = sqlite3.connect(banco)
        self.conexão.row_factory = sqlite3.Row
        if novo:
            self.cria_banco()
        self.carregaTipos()
    def carregaTipos(self):
        for tipo in self.conexão.execute("select * from tipos"):
            id_ = tipo["id"]
            descrição = tipo["descrição"]
            self.tiposTelefone.adiciona(DBTipoTelefone(id_, descrição))
    def cria_banco(self):
        self.conexão.executescript(BANCO)
    def pesquisaNome(self, nome):
        if not isinstance(nome, DBNome):
            raise TypeError("nome deve ser do tipo DBNome")
        achado = self.conexão.execute("""select count(*)
                                         from nomes where nome = ?""",
                                         (nome.nome,)).fetchone()
        if achado[0] > 0:
            return self.carrega_por_nome(nome)
        else:
            return None
    def carrega_por_id(self, id):
        consulta = self.conexão.execute(
                "select * from nomes where id = ?", (id,))
        return self.carrega(consulta.fetchone())
    def carrega_por_nome(self, nome):
        consulta = self.conexão.execute(
                "select * from nomes where nome = ?", (nome.nome,))
```

```
            return self.carrega(consulta.fetchone())
        def carrega(self, consulta):
            if consulta is None:
                return None
            novo = DBDadoAgenda(DBNome(consulta["nome"], consulta["id"]))
            for telefone in self.conexão.execute(
                    "select * from telefones where id_nome = ?", (novo.nome.id,)):
                ntel = DBTelefone(telefone["número"], None,
                                  telefone["id"], telefone["id_nome"])
                for tipo in self.tiposTelefone:
                    if tipo.id == telefone["id_tipo"]:
                        ntel.tipo = tipo
                        break
                novo.telefones.adiciona(ntel)
            return novo
        def lista(self):
            consulta = self.conexão.execute(
                    "select * from nomes order by nome")
            for registro in consulta:
                yield self.carrega(registro)
    def novo(self, registro):
            try:
                cur = self.conexão.cursor()
                cur.execute("insert into nomes(nome) values (?)",
                            (str(registro.nome),))
                registro.nome.id = cur.lastrowid
                for telefone in registro.telefones:
                    cur.execute("""insert into telefones(número,
                                   id_tipo, id_nome) values (?,?,?)""",
                                (telefone.número, telefone.tipo.id,
                                 registro.nome.id))
                    telefone.id = cur.lastrowid
                self.conexão.commit()
            except Exception:
                self.conexão.rollback()
                raise
            finally:
                cur.close()
        def atualiza(self, registro):
            try:
                cur = self.conexão.cursor()
```

```
            cur.execute("update nomes set nome=? where id = ?",
                (str(registro.nome), registro.nome.id))
            for telefone in registro.telefones:
                if telefone.id is None:
                    cur.execute("""insert into telefones(número,
                                   id_tipo, id_nome)
                                   values (?,?,?)""",
                            (telefone.número, telefone.tipo.id,
                            registro.nome.id))
                    telefone.id = cur.lastrowid
                else:
                    cur.execute("""update telefones set número=?,
                                          id_tipo=?, id_nome=?
                                          where id = ?""",
                            (telefone.número, telefone.tipo.id,
                            registro.nome.id, telefone.id))
            for apagado in registro.telefones.apagados:
                cur.execute("delete from telefones where id = ?", (apagado,))
            self.conexão.commit()
            registro.telefones.limpa()
        except Exception:
            self.conexão.rollback()
            raise
        finally:
            cur.close()
    def apaga(self, registro):
        try:
            cur = self.conexão.cursor()
            cur.execute("delete from telefones where id_nome = ?",
                            (registro.nome.id,))
            cur.execute("delete from nomes where id = ?",
                        (registro.nome.id,))
            self.conexão.commit()
        except Exception:
            self.conexão.rollback()
            raise
        finally:
            cur.close()
```

A primeira coisa que a classe `DBAgenda` faz é verificar se o banco de dados já existe. Essa verificação é feita antes do pedido de conexão no método `__init__`.

Para saber se o banco já existe, utilizamos a função `os.path.isfile(banco)`. Caso o arquivo não exista, ele será criado pelo método `cria_banco`, chamado logo após a conexão com o banco de dados. Veja que guardamos o objeto conexão como um atributo de `DBAgenda`.

O método `cria_banco` é muito simples, utilizando o método `executescript` da conexão. Esse método executa vários comandos de uma só vez. Para simplificar a criação do banco, escrevemos o código que cria todas as tabelas e popula a tabela tipos na variável global `BANCO`. A execução de vários comandos só é possível porque eles estão separados por `;`.

O método `carrega_tipos` realiza a leitura de todos os tipos de telefone no banco de dados e os guarda na lista `tiposTelefone`. Observe o cuidado em manter os `id`. Esse valor será utilizado depois para obter o tipo correto de cada telefone.

O método `pesquisa_nome` também traz novidades. Nele, executamos uma consulta usando a função `count(*)`. Se um registro for encontrado, o método `carrega_por_nome` é chamado para transformar o resultado de nossa consulta em uma coleção de objetos, da mesma forma que trabalhamos com a agenda no Capítulo 10. Veja também que utilizamos a cláusula `where` para pesquisar na tabela nomes diretamente, uma vez que não carregamos todos os registros do banco de dados para a memória.

Em `carrega`, o resultado de nossa consulta é transformado em um objeto `DBDadoAgenda`, após criarmos uma instância de `DBNome` com os campos `nome` e `id` vindos de nossa consulta. Esse tipo de acesso é possível, pois registramos `self.conexão.row_factory = sqlite3.Row` no método `__init__`. Uma vez que o nome foi carregado, utilizamos seu `id` como `id_nome` em nossa próxima consulta, que carregará os valores de telefone. Observe o cuidado durante a leitura dos telefones e a transformação do resultado em `DBTelefone`. Como o tipo ainda não foi carregado e temos todos eles em memória, fazemos uma pesquisa em `self.TiposTelefone` para converter o `id` em uma instância de `TipoTelefone`. É esse tipo de mapeamento que não é tão simples de fazer e que é bastante trabalhoso pela grande quantidade de código necessária para mantê-lo. É nessas horas que um ORM ajuda. Uma vez que tudo foi convertido, `novo` é retornado com todos os dados pré-carregados.

O método `lista` executa uma consulta total da tabela nomes. Porém, para evitar a criação de uma grande lista, utilizamos a instrução **`yield`** do Python que retorna cada valor carregado, um de cada vez. Na realidade, o método `lista` retorna um gerador (*generator*, ver Capítulo 8) que pode ser utilizado em um **`for`** do Python. Isso evita termos de carregar e converter todos os valores antes de ter os primeiros resultados na tela.

Em `novo`, convertemos um objeto do tipo `DBDadoAgenda` em registros das tabelas nomes e telefones. Essa conversão é realizada dentro de uma transação, daí o porquê de utilizamos explicitamente um cursor (`cur`). Realizar essa operação dentro de uma transação permitirá melhorar a consistência do banco de dados, pois, se um erro acontecer antes de completarmos todas as operações, o estado do banco de dados será revertido ao estado anterior ao início de nossas operações. Como o `id` de nomes é gerado automaticamente, veja que utilizamos a propriedade `lastrowid` de nosso cursor, logo após a execução do `insert`. Dessa forma, podemos utilizar o novo id para popular a tabela de telefones. Realizamos então o mesmo processo a cada novo telefone, atribuindo o valor de `lastrowid` ao `id` do telefone.

Já o método `atualiza` é bem mais complexo. Durante a atualização de um registro, precisamos atualizar o campo `nome` na tabela de nomes. Em nossa solução caseira, não temos como saber se nome foi alterado ou não. Poderíamos alterar a classe para saber quando o nome foi alterado e executar o `update` apenas quando necessário. Essa é outra característica das bibliotecas de ORM que trazem esse tipo de funcionalidade em suas classes. Para mantermos a agenda o mais simples possível, `nome` sempre será atualizado. Para cada `telefone`, fazemos a verificação do valor de `telefone.id`. Se um telefone já possui um valor em `id`, provavelmente ele já está registrado no banco de dados e precisa ser atualizado. Caso ainda não possua um valor em `id`, provavelmente se trata de um telefone inserido durante a alteração. Dessa forma, escolhemos entre fazer um `insert` ou um `update` na tabela telefones.

Por último, verificamos se a lista de apagados possui uma lista de `id` a apagar. Essa lista foi construída pela classe `DBListaÚnica`, em que guardamos o `id` de cada elemento apagado. Essa mudança foi necessária, pois, diferente dos novos registros e dos registros alterados, os registros apagados são removidos de nossa lista. Se não mantivermos a lista dos `id` apagados, ficaremos sem saber quais telefones foram removidos, e nosso banco ficará inconsistente. Para cada `id` na lista de apagados, executamos um `delete`. Logo após, terminamos a transação com um `commit` para nos assegurarmos de que todas as operações foram registradas no banco de dados e, só então, apagamos a lista de telefones removidos com o método `limpa`.

O método `apaga` é um pouco mais simples. Nele, utilizamos a mesma estrutura de proteção e uma transação. O interessante é que, primeiro, apagamos os telefones e, depois, os nomes. Da forma que geramos nosso banco de dados, essa ordem não importa, uma vez que não estamos utilizando os recursos de integridade referencial do banco.

O programa da agenda é apresentado por inteiro:

```
# Programa 11.7 - Agenda com banco de dados completo
import sys
import sqlite3
import os.path
from functools import total_ordering
BANCO = """
create table tipos(id integer primary key autoincrement,
                   descrição text);
create table nomes(id integer primary key autoincrement,
                   nome text);
create table telefones(id integer primary key autoincrement,
                   id_nome integer,
                   número text,
                   id_tipo integer);
insert into tipos(descrição) values ("Celular");
insert into tipos(descrição) values ("Fixo");
insert into tipos(descrição) values ("Fax");
insert into tipos(descrição) values ("Casa");
insert into tipos(descrição) values ("Trabalho");
"""

def nulo_ou_vazio(texto):
    return texto is None or not texto.strip()
def valida_faixa_inteiro(pergunta, início, fim, padrão=None):
    while True:
        try:
            entrada = input(pergunta)
            if nulo_ou_vazio(entrada) and padrão is not None:
                entrada = padrão
            valor = int(entrada)
            if início <= valor <= fim:
                return valor
        except ValueError:
            print(f"Valor inválido, favor digitar entre {início} e {fim}")
def valida_faixa_inteiro_ou_branco(pergunta, início, fim):
    while True:
        try:
            entrada = input(pergunta)
```

```
            if nulo_ou_vazio(entrada):
                return None
            valor = int(entrada)
            if início <= valor <= fim:
                return valor
        except ValueError:
            print(f"Valor inválido, favor digitar entre {início} e {fim}")
class ListaÚnica:
    def __init__(self, elem_class):
        self.lista = []
        self.elem_class = elem_class
    def __len__(self):
        return len(self.lista)
    def __iter__(self):
        return iter(self.lista)
    def __getitem__(self, p):
        return self.lista[p]
    def indiceVálido(self, i):
        return i >= 0 and i < len(self.lista)
    def adiciona(self, elem):
        if self.pesquisa(elem) == -1:
            self.lista.append(elem)
    def remove(self, elem):
        self.lista.remove(elem)
    def pesquisa(self, elem):
        self.verifica_tipo(elem)
        try:
            return self.lista.index(elem)
        except ValueError:
            return -1
    def verifica_tipo(self, elem):
        if not isinstance(type(elem), self.elem_class):
            raise TypeError("Tipo inválido")
    def ordena(self, chave=None):
        self.lista.sort(key=chave)
class DBListaÚnica(ListaÚnica):
    def __init__(self, elem_class):
        super().__init__(elem_class)
        self.apagados = []
```

```
        def remove(self, elem):
            if elem.id is not None:
                self.apagados.append(elem.id)
            super().remove(elem)
        def limpa(self):
            self.apagados = []
    @total_ordering
    class Nome:
        def __init__(self, nome):
            self.nome = nome
        def __str__(self):
            return self.nome
        def __repr__(self):
            return f"<Classe {type(self).__name__} em 0x{id(self):x} Nome: {self.__
        nome} Chave: {self.__chave }>"
        def __eq__(self, outro):
            return self.nome == outro.nome
        def __lt__(self, outro):
            return self.nome < outro.nome
        @property
        def nome(self):
            return self.__nome
        @nome.setter
        def nome(self, valor):
            if nulo_ou_vazio(valor):
                raise ValueError("Nome não pode ser nulo nem em branco")
            self.__nome = valor
            self.__chave = Nome.CriaChave(valor)
        @property
        def chave(self):
            return self.__chave
        @staticmethod
        def CriaChave(nome):
            return nome.strip().lower()
    class DBNome(Nome):
        def __init__(self, nome, id_=None):
            super().__init__(nome)
            self.id = id_
    @total_ordering
```

```
class TipoTelefone:
    def __init__(self, tipo):
        self.tipo = tipo
    def __str__(self):
        return "({0})".format(self.tipo)
    def __eq__(self, outro):
        if outro is None:
            return False
        return self.tipo == outro.tipo
    def __lt__(self, outro):
        return self.tipo < outro.tipo
class DBTipoTelefone(TipoTelefone):
    def __init__(self, id_, tipo):
        super().__init__(tipo)
        self.id = id_
class Telefone:
    def __init__(self, número, tipo=None):
        self.número = número
        self.tipo = tipo
    def __str__(self):
        if self.tipo is not None:
            tipo = self.tipo
        else:
            tipo = ""
        return f"{self.número} {tipo}"
    def __eq__(self, outro):
        return self.número == outro.número and (
               (self.tipo == outro.tipo) or (
                self.tipo is None or outro.tipo is None))
    @property
    def número(self):
        return self.__número
    @número.setter
    def número(self, valor):
        if nulo_ou_vazio(valor):
            raise ValueError("Número não pode ser None ou em branco")
        self.__número = valor
```

```
class DBTelefone(Telefone):
    def __init__(self, número, tipo=None, id_=None, id_nome=None):
        super().__init__(número, tipo)
        self.id = id_
        self.id_nome = id_nome
class DBTelefones(DBListaÚnica):
    def __init__(self):
        super().__init__(DBTelefone)
class DBTiposTelefone(ListaÚnica):
    def __init__(self):
        super().__init__(DBTipoTelefone)
class DBDadoAgenda:
    def __init__(self, nome):
        self.nome = nome
        self.telefones = DBTelefones()
    @property
    def nome(self):
        return self.__nome
    @nome.setter
    def nome(self, valor):
        if not isinstance(type(valor), DBNome):
            raise TypeError("nome deve ser uma instância da classe DBNome")
        self.__nome = valor
    def pesquisaTelefone(self, telefone):
        posição = self.telefones.pesquisa(DBTelefone(telefone))
        if posição == -1:
            return None
        else:
            return self.telefones[posição]
class DBAgenda:
    def __init__(self, banco):
        self.tiposTelefone = DBTiposTelefone()
        self.banco = banco
        novo = not os.path.isfile(banco)
        self.conexão = sqlite3.connect(banco)
        self.conexão.row_factory = sqlite3.Row
        if novo:
            self.cria_banco()
        self.carregaTipos()
```

```
    def carregaTipos(self):
        for tipo in self.conexão.execute("select * from tipos"):
            id_ = tipo["id"]
            descrição = tipo["descrição"]
            self.tiposTelefone.adiciona(DBTipoTelefone(id_, descrição))
    def cria_banco(self):
        self.conexão.executescript(BANCO)
    def pesquisaNome(self, nome):
        if not isinstance(nome, DBNome):
            raise TypeError("nome deve ser do tipo DBNome")
        achado = self.conexão.execute("""select count(*)
                                         from nomes where nome = ?""",
                                      (nome.nome,)).fetchone()
        if achado[0] > 0:
            return self.carrega_por_nome(nome)
        else:
            return None
    def carrega_por_id(self, id):
        consulta = self.conexão.execute(
                "select * from nomes where id = ?", (id,))
        return self.carrega(consulta.fetchone())
    def carrega_por_nome(self, nome):
        consulta = self.conexão.execute(
                "select * from nomes where nome = ?", (nome.nome,))
        return self.carrega(consulta.fetchone())
    def carrega(self, consulta):
        if consulta is None:
            return None
        novo = DBDadoAgenda(DBNome(consulta["nome"], consulta["id"]))
        for telefone in self.conexão.execute(
                "select * from telefones where id_nome = ?",
                (novo.nome.id,)):
            ntel = DBTelefone(telefone["número"], None,
                              telefone["id"], telefone["id_nome"])
            for tipo in self.tiposTelefone:
                if tipo.id == telefone["id_tipo"]:
                    ntel.tipo = tipo
                    break
            novo.telefones.adiciona(ntel)
```

```
        return novo
    def lista(self):
        consulta = self.conexão.execute(
                "select * from nomes order by nome")
        for registro in consulta:
            yield self.carrega(registro)
    def novo(self, registro):
        try:
            cur = self.conexão.cursor()
            cur.execute("insert into nomes(nome) values (?)",
                        (str(registro.nome),))
            registro.nome.id = cur.lastrowid
            for telefone in registro.telefones:
                cur.execute("""insert into telefones(número,
                               id_tipo, id_nome) values (?,?,?)""",
                            (telefone.número, telefone.tipo.id,
                             registro.nome.id))
                telefone.id = cur.lastrowid
            self.conexão.commit()
        except Exception:
            self.conexão.rollback()
            raise
        finally:
            cur.close()
    def atualiza(self, registro):
        try:
            cur = self.conexão.cursor()
            cur.execute("update nomes set nome=? where id = ?",
                        (str(registro.nome), registro.nome.id))
            for telefone in registro.telefones:
                if telefone.id is None:
                    cur.execute("""insert into telefones(número,
                                   id_tipo, id_nome)
                                   values (?,?,?)""",
                                (telefone.número, telefone.tipo.id, registro.nome.id))
                    telefone.id = cur.lastrowid
                else:
                    cur.execute("""update telefones set número=?,
                                   id_tipo=?, id_nome=?
                                   where id = ?""",
```

```
                                    (telefone.número, telefone.tipo.id,
                                     registro.nome.id, telefone.id))
                for apagado in registro.telefones.apagados:
                    cur.execute("delete from telefones where id = ?", (apagado,))
                self.conexão.commit()
                registro.telefones.limpa()
            except Exception:
                self.conexão.rollback()
                raise
            finally:
                cur.close()
        def apaga(self, registro):
            try:
                cur = self.conexão.cursor()
                cur.execute("delete from telefones where id_nome = ?", (registro.nome.
    id,))
                cur.execute("delete from nomes where id = ?", (registro.nome.id,))
                self.conexão.commit()
            except Exception:
                self.conexão.rollback()
                raise
            finally:
                cur.close()
    class Menu:
        def __init__(self):
            self.opções = [["Sair", None]]
        def adicionaopção(self, nome, função):
            self.opções.append([nome, função])
        def exibe(self):
            print("====")
            print("Menu")
            print("====\n")
            for i, opção in enumerate(self.opções):
                print(f"[{i}] - {opção[0]}")
            print()
        def execute(self):
            while True:
                self.exibe()
```

```
            escolha = valida_faixa_inteiro("Escolha uma opção: ", 0, len(self.opções)-1)
            if escolha == 0:
                break
            self.opções[escolha][1]()
class AppAgenda:
    @staticmethod
    def pede_nome():
        return input("Nome: ")
    @staticmethod
    def pede_telefone():
        return input("Telefone: ")
    @staticmethod
    def mostra_dados(dados):
        print(f"Nome: {dados.nome}")
        for telefone in dados.telefones:
            print(f"Telefone: {telefone}")
        print()
    @staticmethod
    def mostra_dados_telefone(dados):
        print(f"Nome: {dados.nome}")
        for i, telefone in enumerate(dados.telefones):
            print(f"{i} - Telefone: {telefone}")
        print()
    def __init__(self, banco):
        self.agenda = DBAgenda(banco)
        self.menu = Menu()
        self.menu.adicionaopção("Novo", self.novo)
        self.menu.adicionaopção("Altera", self.altera)
        self.menu.adicionaopção("Apaga", self.apaga)
        self.menu.adicionaopção("Lista", self.lista)
        self.ultimo_nome = None
    def pede_tipo_telefone(self, padrão=None):
        for i, tipo in enumerate(self.agenda.tiposTelefone):
            print(f" {i} - {tipo} ", end=None)
        t = valida_faixa_inteiro(
            "Tipo: ", 0,
            len(self.agenda.tiposTelefone)-1, padrão)
        return self.agenda.tiposTelefone[t]
```

```
    def pesquisa(self, nome):
        if isinstance(type(nome), str):
            nome = DBNome(nome)
        dado = self.agenda.pesquisaNome(nome)
        return dado
    def novo(self):
        novo = AppAgenda.pede_nome()
        if nulo_ou_vazio(novo):
            return
        nome = DBNome(novo)
        if self.pesquisa(nome) is not None:
            print("Nome já existe!")
            return
        registro = DBDadoAgenda(nome)
        self.menu_telefones(registro)
        self.agenda.novo(registro)
    def apaga(self):
        nome = AppAgenda.pede_nome()
        if nulo_ou_vazio(nome):
            return
        p = self.pesquisa(nome)
        if p is not None:
            self.agenda.apaga(p)
        else:
            print("Nome não encontrado.")
    def altera(self):
        nome = AppAgenda.pede_nome()
        if nulo_ou_vazio(nome):
            return
        p = self.pesquisa(nome)
        if p is not None:
            print("\nEncontrado:\n")
            AppAgenda.mostra_dados(p)
            print("Digite enter caso não queira alterar o nome")
            novo = AppAgenda.pede_nome()
            if not nulo_ou_vazio(novo):
                p.nome.nome = novo
            self.menu_telefones(p)
```

```
            self.agenda.atualiza(p)
        else:
            print("Nome não encontrado!")
    def menu_telefones(self, dados):
        while True:
            print("\nEditando telefones\n")
            AppAgenda.mostra_dados_telefone(dados)
            if len(dados.telefones) > 0:
                print("\n[A] - alterar\n[D] - apagar\n", end="")
            print("[N] - novo\n[S] - sair\n")
            operação = input("Escolha uma operação: ")
            operação = operação.lower()
            if operação not in ["a", "d", "n", "s"]:
                print("Operação inválida. Digite A, D, N ou S")
                continue
            if operação == 'a' and len(dados.telefones) > 0:
                self.altera_telefones(dados)
            elif operação == 'd' and len(dados.telefones) > 0:
                self.apaga_telefone(dados)
            elif operação == 'n':
                self.novo_telefone(dados)
            elif operação == "s":
                break
    def novo_telefone(self, dados):
        telefone = AppAgenda.pede_telefone()
        if nulo_ou_vazio(telefone):
            return
        if dados.pesquisaTelefone(telefone) is not None:
            print("Telefone já existe")
        tipo = self.pede_tipo_telefone()
        dados.telefones.adiciona(DBTelefone(telefone, tipo))
    def apaga_telefone(self, dados):
        t = valida_faixa_inteiro_ou_branco(
            "Digite a poisção do número a apagar, enter para sair: ",
            0, len(dados.telefones)-1)
        if t is None:
            return
        dados.telefones.remove(dados.telefones[t])
```

```
    def altera_telefones(self,dados):
        t = valida_faixa_inteiro_ou_branco(
            "Digite a poisção do número a alterar, enter para sair: ",
            0, len(dados.telefones)-1)
        if t is None:
            return
        telefone = dados.telefones[t]
        print(f"Telefone: {telefone}")
        print("Digite enter caso não queira alterar o número")
        novotelefone = AppAgenda.pede_telefone()
        if not nulo_ou_vazio(novotelefone):
            telefone.número = novotelefone
        print("Digite enter caso não queira alterar o tipo")
        telefone.tipo = self.pede_tipo_telefone(
            self.agenda.tiposTelefone.pesquisa(telefone.tipo))
    def lista(self):
        print("\nAgenda")
        print("-" * 60)
        for e in self.agenda.lista():
            AppAgenda.mostra_dados(e)
        print("-" * 60)
    def execute(self):
        self.menu.execute()
if __name__ == "__main__":
    if len(sys.argv) > 1:
        app = AppAgenda(sys.argv[1])
        app.execute()
    else:
        print("Erro: nome do banco de dados não informado")
        print("       agenda.py nome_do_banco")
```

Na classe `AppAgenda`, modificamos as opções do menu e os tipos usados nas pesquisas. Veja que, com a utilização da classe `DBAgenda`, conseguimos isolar as operações de banco de dados da classe `AppAgenda`. Como não temos operações de leitura e gravação, o nome do banco de dados deve ser passado obrigatoriamente na linha de comando. Uma mensagem de erro será exibida caso você se esqueça desse detalhe.

# CAPÍTULO 12

# Próximos passos

A programação é um longo caminho que você começou a percorrer. Neste livro, mostramos as técnicas básicas para que você continue seus estudos. A programação de computadores é uma área de conhecimento muito grande e rica. Este livro apresentou uma introdução à programação utilizando a linguagem Python. Os próximos passos dependem de sua área de interesse. Você não precisa estudar todos os tópicos ou mesmo seguir a ordem em que são apresentados neste capítulo. Vamos listar alguns tópicos.

## 12.1 Programação funcional

A programação funcional é um paradigma de programação diferente do que aprendemos aqui com Python. Embora Python tenha recursos de uma linguagem funcional, permitindo misturar os dois paradigmas em um só programa. Para enriquecer sua experiência em programação, estudar uma linguagem puramente funcional, como LISP, Scheme ou F#, é realmente interessante, sendo uma área que deve popularizar-se cada vez mais. Estudar uma nova forma de organizar seus programas ajudará a entender melhor vários conceitos que você já aprendeu e a modificar a forma que você encara a programação.

Atualmente, diversas linguagens são chamadas de híbridas ou multiparadigmas, pois misturam ou possibilitam a utilização de recursos de programação funcional e imperativa. Python é uma delas. Além disso, algumas propriedades da programação funcional são excelentes para resolver ou abordar problemas de concorrência, cada vez mais comuns hoje em dia.

O livro *SICP – Structure and Interpretation of Computer Programs*, de Harold Abelson e Gerald Jay Sussman, pode ser lido online no site `https://mitpress.mit.edu/sites/default/files/sicp/index.html`. O livro é usado em diversos cursos de computação e é considerado um dos melhores disponíveis.

## 12.2 Algoritmos

Até agora, utilizamos o básico da programação para resolver nossos problemas. À medida que você for ganhando experiência, terá de estudar novos algoritmos e entender algoritmos que já utiliza hoje, sem perceber. Um exemplo é a compreensão de técnicas de espalhamento (*hashes*), que utilizamos em Python com dicionários, ou como as listas funcionam internamente. Quanto maior seus programas, mais importante entender como cada algoritmo funciona, conhecer suas aplicações e limitações.

A linguagem C também é recomendada para o estudo de algoritmos e estruturas de dados. Ao contrário de Python, em C você está sozinho e ela vem quase sem baterias. A vantagem de estudar e aprender a programar em C é conhecer os detalhes de cada implementação e controlar cada passo de seus programas.

Um livro recomendado para estudos aprofundados é *Algoritmos: teoria e prática*, de Thomas H. Cormen. Embora a leitura não seja das mais fáceis, seu conteúdo é esclarecedor. Não leia esse livro antes de programar em pelo menos duas linguagens de programação. Sua leitura é recomendada para pessoas que desejem realmente se aprofundar no assunto, normalmente cursando ciência da computação.

## 12.3 Jogos

A programação de jogos é uma das áreas mais recompensadoras e inspiradoras. Além disso, o que você aprende programando jogos serve para solucionar diversos outros problemas do dia a dia. Python apresenta diversas bibliotecas externas prontas para a criação de jogos em 2D ou 3D. Essas bibliotecas incluem a manipulação de imagens, sons, controle do tempo e mesmo de arquivos.

- **Pygame** (`http://www.pygame.org`): biblioteca multimídia fácil de aprender. Permite a manipulação de superfícies de desenho e a criação de jogos com gráficos em 2D e som. Se você tiver curiosidade de saber como criar jogos simples, visite o projeto Invasores (`https://github.com/lskbr/invasores`). O projeto foi inteiramente desenvolvido em Python e é open source. Você pode baixar e estudar o código-fonte, fazer modificações e testar. Tudo escrito em Python.
- **Panda 3D** (`http://www.panda3d.org`): biblioteca gráfica, desenvolvida pelos estúdios Disney, open source e supercompleta. Com Panda 3D você pode criar jogos profissionais, e o melhor de tudo: usando Python. A biblioteca é poderosa e complexa, tendo sido utilizada em jogos comerciais. Não se esqueça de revisar álgebra linear e estudar gráficos tridimensionais antes de se aventurar na documentação (em inglês).

- **PyOgre** (`http://www.ogre3d.org/tikiwiki/PyOgre`): outra biblioteca ou kit de desenvolvimento de jogos 3D. A PyOgre é também poderosa e complexa.

Se você começar a estudar essas bibliotecas, encare essa tarefa como um projeto de pelo menos um ano. Essas bibliotecas são grandes e permitem a criação de aplicativos multimídia, além de jogos, claro. O ideal é realizar esse estudo em grupo: visite o site `http://www.unidev.com.br`.

## 12.4 Orientação a objetos

No Capítulo 10, apenas introduzimos a programação orientada a objetos. Como já dito, esse é um assunto realmente extenso que vale a pena ser estudado. Python é uma linguagem que implementa quase todos os conceitos de orientação a objeto, mas você deve aprender a dominá-los. Estude poliformismo, interfaces e padrões de projeto (*design patterns*).

Um livro que ajuda a decifrar a orientação a objetos é *Head First Design Patterns*, de Eric T. Freeman, Elisabeth Robson, Bert Bates e Kathy Sierra. O livro apresenta os conceitos básicos e os principais padrões de projeto de forma simples e divertida, com diversas histórias e gráficos.

Um livro clássico sobre padrões de projeto é *Design Patterns: Elements of Reusable Object-Oriented Software*, de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. O livro apresenta um conteúdo denso em formato de referência. Não é indicado para uma introdução a padrões de projeto, mas você deve lê-lo um dia.

## 12.5 Banco de dados

Embora tenhamos abordado banco de dados no Capítulo 11, o assunto é extenso.

Depois de ler a documentação do módulo sqlite3, você pode aventurar-se com bibliotecas mais modernas que isolam seus programas do banco de dados em si, tornando a manipulação de seus dados quase transparente. Uma dica é a biblioteca SQLAlchemy (`http://www.sqlalchemy.org`); veja também o Pony (`https://ponyorm.com`).

Existem também diversos bancos de dados modernos que não usam SQL, os chamados bancos NoSQL. Entre eles, destacam-se Redis (`https://redis.io`), MongoDB (`https://www.mongodb.com`), Cassandra (`http://cassandra.apache.org`), Amazon DynamoDB (`https://github.com/pynamodb/PynamoDB`).

Os bancos de dados clássicos que você deve conhecer são o PostgreSQL (`https://www.postgresql.org`) e o MySQL (`https://www.mysql.com`).

## 12.6 Sistemas web

Um dos assuntos mais quentes hoje em dia é a programação de sistemas web. Esses sistemas também podem ser escritos em Python. No entanto, a comunicação de um programa com o servidor de páginas web é repleta de detalhes e torna seus programas repetitivos e trabalhosos de escrever. Se você deseja aprender a escrever, ou se interessa por sistemas web, estude o Django (`http://www.djangoproject.com`). O Django estrutura seus sistemas para que se tornem fáceis de escrever e modificar. Além disso, as bibliotecas do Django já realizam as tarefas repetitivas de comunicação com o servidor web e mesmo com o banco de dados.

Outro framework muito utilizado para desenvolver sistemas web é o Flask (`http://flask.pocoo.org`).

Sistemas em cloud computing também podem ser escritos em Python. Um framework muito interessante é o Chalice, desenvolvido pela própria Amazon Web Services (`https://github.com/aws/chalice`), e capaz de criar funções com AWS Lambda, uma forma de FaaS (*Function as a Service*).

## 12.7 Ciência de dados e inteligência artificial

As áreas de ciências de dados e inteligência artificial têm crescido enormemente e usando Python. A razão dessa adoção de linguagem nesses campos é a facilidade de programação e a quantidade de bibliotecas disponíveis.

Para aplicações científicas ou de cálculo intenso, não se esqueça de olhar os módulos NumPy (`http://www.numpy.org`) e Scipy (`http://www.scipy.org`). Se você precisar trabalhar com gráficos (coluna, barra, torta, superfície e muitos outros), procure o módulo matplotlib (`https://matplotlib.org`). A biblioteca Pandas também é muito utilizada para trabalhar com análise de dados (`https://pandas.pydata.org`).

Bibliotecas como TensorFlow (`https://www.tensorflow.org`), Keras (`https://keras.io`), Scikit-learn (`http://scikit-learn.org`) são usadas para machine learning e deep learning.

## 12.8 Outras bibliotecas Python

Além de Python vir com "baterias incluídas", você encontra na internet diversas bibliotecas ou mais baterias para fazer praticamente tudo. Um bom site para procurar um módulo é o Python Package Index – PyPI (`https://pypi.org`).

Outro projeto que não pode deixar de ser visitado é o Pycairo (`http://cairographics.org/pycairo`), que permite o desenho de gráficos com recursos avançados, como suavização de curvas e transparências. Para trabalhar com imagens (PNG, JPG etc.), o módulo PILLOW, ou fork do *Python Image Library* (`https://python-pillow.org`), é referência.

Se você deseja adicionar cores a seu console, imprimindo mensagens com cores diferentes, limpando a tela e mesmo trabalhando com o teclado de forma mais direta, visite a ColorConsole (`https://github.com/lskbr/colorconsole`).

## 12.8 Listas de discussão

Uma forma de não ficar isolado é participar de listas de discussão. Para se aprofundar e continuar a estudar Python, inscreva-se na lista python-brasil (`https://python.org.br/lista-de-discussoes`). Essa lista é muito ativa e conta com uma comunidade participativa que acolhe dúvidas básicas e avançadas. Antes de perguntar, leia os guias disponíveis no site e as regras de como fazer perguntas. O histórico da lista também traz milhares de perguntas e respostas que você pode pesquisar. A wiki é rica em exemplos de problemas e soluções escritas em Python e disponíveis em português.

Já para quem estiver estudando Django, a lista de discussão é django-brasil (`http://groups.google.com/group/django-brasil/?pli=1`).

Nos últimos anos, as listas de discussão começaram a ser trocadas por programas como o Telegram. Existem vários grupos e subgrupos de Python no Telegram. Os principais são o da Python Brasil (`https://t.me/pythonbr`) e o PyCoding (`https://t.me/PyCoding`). Você pode enviar uma mensagem para o autor utilizando o endereço `@lskbr`.

# APÊNDICE A

# Mensagens de erro

Erros sempre podem ocorrer. Ora digitamos algo errado, ora nos esquecemos de digitar algo importante.

O interpretador Python informa erros por meio de mensagens que indicam o tipo do erro, assim como onde ele ocorreu (arquivo e linha). As seções deste apêndice mostram a listagem de um programa e a mensagem de erro respectiva. Lembre-se de que o interpretador segue regras como qualquer programa e que, às vezes, você pode ter mensagens causadas por erros em outras linhas de seu programa. Utilize as mensagens de erro como um bom palpite do que ocorreu. O que elas realmente indicam é onde, em seu programa, o interpretador foi interrompido e a causa dessa interrupção. É investigando esse palpite que você encontrará o verdadeiro erro.

## A.1 SyntaxError

Um dos tipos mais comuns. Um erro de sintaxe acontece quando o interpretador não consegue ler o que você escreveu. Em outras palavras, seu programa está mal formado, normalmente com erros de digitação ou símbolos esquecidos.

```
a = "5
  File "erros/sintax.py", line 1
    a = "5
         ^
SyntaxError: EOL while scanning string literal
```

No exemplo anterior, a linha foi terminada sem fechar aspas.

```
for e in [1,2,3]
    print(e)
  File "erros/sintax2.py", line 1
```

```
    for e in [1,2,3]
                  ^
SyntaxError: invalid syntax
```

Todas as linhas com **for**, **while**, **if** e **else** devem ser encerradas com o símbolo de : para indicar o início de um novo bloco.

```
a = 10
print a
  File "erros/sintax4.py", line 2
    print a
          ^
SyntaxError: invalid syntax
```

Aqui a função **print** foi utilizada, mas observe que o parâmetro a não foi escrito entre parênteses.

Observe que um circunflexo é exibido na coluna em que o interpretador considera que o erro pode ter acontecido. Essa indicação é apenas uma dica, devendo ser investigada caso a caso. Alguns erros podem fazer com que o interpretador se perca, indicando o lugar errado. Sempre leia a linha indicada na mensagem de erro, mas não se esqueça de também olhar as linhas anteriores, caso não ache o erro.

Sempre que ocorrer um erro de sintaxe, verifique:

1. A linha onde ocorreu um erro (*line*).
2. Se você fechou todas as aspas que abriu, o mesmo valendo para parênteses.
3. Os dois pontos após o **while**, **for**, **if**, **else**, definições de função, métodos e classes.
4. Se você não trocou letras minúsculas por maiúsculas e vice-versa.
5. Se você digitou corretamente todos os nomes.

## A.2 IdentationError

```
x = 0
while x < 10:
    print(x)
  x = x + 1
  File "erros/sintax5.py", line 4
    x = x + 1
        ^
IndentationError: unindent does not match any outer indentation level
```

Nesse caso, observe que a linha `x = x + 1` não está alinhada nem com o bloco do **`print`** nem com o **`while`**. Todas as linhas de um mesmo bloco devem ser alinhadas na mesma coluna.

Uma variação muito difícil de perceber desse erro é quando misturamos espaços em branco com tabulações (tabs). Configure seu editor de textos para substituir tabs por espaços em branco ou vice-versa. Jamais misture tabulações e espaços em branco em seus programas em Python.

## A.3 KeyError

```
>>> mensalidades = {"carro": 500, "casa": 1500}
>>> print(mensalidades["seguro"])
Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <module>
    print(mensalidades["seguro"])
KeyError: 'seguro'
```

A exceção `KeyError` ocorre quando acessamos ou tentamos acessar um dicionário usando uma chave que não existe. No exemplo anterior, o dicionário `mensalidades` contém as chaves carro e casa. Na linha da função **`print`**, tentamos acessar `mensalidades["seguro"]`. Nesse caso, `"seguro"` é a chave que causa a mensagem de erro, uma vez que ela não pertence ao dicionário. Atenção ao trabalhar com chaves string, pois Python diferencia letras minúsculas de maiúsculas.

## A.4 NameError

```
while x < 0:
    print(x)
    x = x + 1
Traceback (most recent call last):
  File "erros/sintax3.py", line 1, in <module>
    while x < 0:
NameError: name 'x' is not defined
```

Aqui a variável `x` é utilizada em **`while`**, mesmo antes de ser iniciada. Toda variável em Python precisa ser inicializada antes de ser utilizada. Lembre-se de que, para inicializarmos uma variável, devemos atribuir um valor inicial. No caso, poderíamos escrever `x = 0` antes da linha de **`while`** para inicializar a variável `x` com zero, resolvendo o erro.

Ao receber esse erro, verifique também se escreveu corretamente o nome da variável. Lembre-se de que o nome de uma variável, como qualquer identificador em Python, leva em consideração variações como letras minúsculas e maiúsculas. Por exemplo, `X = 0` não resolveria o erro, pois `x` e `X` são variáveis diferentes. Não se esqueça também de digitar corretamente os acentos, pois nomes acentuados são também diferentes de nomes sem acentos. Assim, `posicao` é diferente de `posição`.

## A.5 ValueError

A exceção `ValueError` pode acontecer por diversas causas. Uma delas é a impossibilidade de converter um valor com as funções **`int`** ou **`float`**:

```
>>> int("abc")
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    int("abc")
ValueError: invalid literal for int() with base 10: 'abc'
>>> float("maria")
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    float("maria")
ValueError: could not convert string to float: maria
```

Ela pode ocorrer se, por exemplo, o valor retornado pela função **`input`** for inválido.

Essa exceção também ocorre quando procuramos uma string que não existe, como mostrado a seguir.

```
>>> s = "Alô mundo"
>>> s.index("rei")
Traceback (most recent call last):
  File "<pyshell#64>", line 1, in <module>
    s.index("rei")
ValueError: substring not fo-nd
```

## A.6 TypeError

Essa exceção acontece se tentamos chamar uma função usando mais parâmetros do que ela recebe. No exemplo a seguir, a função **`float`** foi chamada com dois parâmetros: 35 e 4. Lembre-se de que números em Python devem ser escritos no formato americano, separando a parte inteira da parte decimal de um número

com ponto e não com vírgula. Esse também é um exemplo de que um erro pode ser apresentado como outro erro, exigindo que você sempre interprete a mensagem de forma a saber o que realmente aconteceu.

```
>>> float(35, 4)
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    float(35, 4)
TypeError: float() takes at most 1 argument (2 given)
```

`TypeError` também ocorre quando trocamos o tipo de um índice. No exemplo a seguir, tentamos utilizar a string "marrom" como índice de uma lista. Lembre-se de que listas só podem ser indexadas por números inteiros. Dicionários, por sua vez, permitem índices do tipo string.

```
>>> s["amarelo", "vermelho", "verde"]
>>> s["marrom"]
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    s["marrom"]
TypeError: string indices must be integers
```

## A.7 IndexError

`IndexError` indica que um valor inválido de índice foi utilizado. No exemplo a seguir, a string `s` contém apenas cinco elementos, podendo ter índices de 0 a 4.

```
>>> s = "ABCDE"
>>> s[10]
Traceback (most recent call last):
  File "<pyshell#66>", line 1, in <module>
    s[10]
IndexError: string index out of range
```

## A.8 TabError

Esse erro acontece quando misturamos tabs (caractere gerado quando você pressiona a tecla tab) e espaços para fazer os recuos e avanços em nossos programas. Esse erro é praticamente invisível, mas pode ser corrigido por seu editor de textos. Procure uma opção que converta todos os caracteres de tab por espaços para resolvê-lo.

APÊNDICE B

# Adaptações para outras versões do Python

## B.1 Python 2.7

Este livro foi escrito visando à versão 3.x da linguagem. Se você não puder instalar a versão mais nova do interpretador, precisará converter as listagens para Python 2.7:

1. Python 2.7 não suporta nomes de variáveis com acentos. Remova todos os acentos dos nomes de função, classes e variáveis.
2. Python 2.7 não utiliza a codificação utf-8 por padrão, adicione a seguinte linha no início de seus programas para suportar utf-8:

   ```
   # -*- coding: utf-8 -*-
   ```

3. Python 2.7 não tem a função **print**, mas o comando **print**. Para utilizar **print** como função, adicione nas primeiras linhas de seus programas:

   ```
   from __future__ import print_function
   ```

4. A sintaxe do **super()** é diferente em Python 2.7, você deve passar o nome da classe como parâmetro:

   Python 3.x:

   ```
   super().__init__()
   ```

   Python 2.7:

   ```
   super(NomeDaClasse, self).__init__()
   ```

5. Suas classes devem derivar de **object**:

   Python 3.x:

   ```
   class Nome:
   ```

   Python 2.x:

   ```
   class Nome(object)
   ```

6. Python 2.7 possui dois tipos de string: `str` e `unicode`. A conversão não é tão simples, pois algumas funções esperam o tipo `str` (que não é unicode, mas semelhante ao tipo `byte` do Python 3), que suporta apenas caracteres sem acentos, usando a tabela ASCII ou strings.

   Em alguns casos, basta prefixar suas strings com `u` para ter o mesmo comportamento que em Python 3:

   Python 2.7:

   ```
   u"String unicode"
   "String normal, ASCII"
   ```

7. A função **`input`** funciona de forma diferente

   Em Python 2.7, a função **`input`** interpreta o valor digitado como código Python! Dessa forma, para ter o mesmo comportamento e retornar uma string, utilize a função `raw_input`.

## B.2 Python 3.5 ou anterior

Se você não puder instalar a versão 3.6 ou superior do Python, todo código que utiliza f-strings deverá ser convertido para usar o método format:

- Python >= 3.6:

  ```
  f"{nome} - {endereço}"
  ```

- Python <= 3.5:

  ```
  "{0} - {1}".format(nome, endereço)
  ```

  ou

  ```
  "{nome} - {endereço}".format(nome=nome, endereço=endereço)
  ```

# Referências

Python Foundation. *Python 3.1.2 Tutorial*. 2010. Disponível em: `http://docs.python.org/3/tutorial`.

Summerfield, M. *Programming in Python 3 – A Complete Introduction to the Python Language*. 2 ed. Addison-Wesley, 2010.

Pilgrim, M. *Dive into Python 3. 2 ed.* Apress, 2009.

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2002.

## Símbolos



## A



## B



## C



## D



## E



## F

## T



## U



## V



## Y